

# Rethinking Solvers for Extreme Scale Architectures

William Gropp

[www.cs.illinois.edu/~wgropp](http://www.cs.illinois.edu/~wgropp)

[parallel.illinois.edu](http://parallel.illinois.edu)



# Performance, then Productivity

---

- Note the “then” – not “instead of”
  - ◆ For “easier” problems, it is correct to invert these
- For the very hardest problems, we must focus on getting the best performance possible
  - ◆ Rely on other approaches to manage the complexity of the codes
  - ◆ Performance can be understood and engineered (note I did not say predicted)
- We need to start now, to get practice
  - ◆ “Vector” instructions, GPUs, extreme scale networks
  - ◆ Because Exascale platforms will be even more complex and harder to use effectively



# Exascale Directions

- Exascale systems are likely to have
  - ◆ Extreme power constraints, leading to
    - Clock Rates similar to today's systems
    - A wide-diversity of simple computing elements (simple for hardware but complex for software)
    - Memory per core and per FLOP will be much smaller
    - Moving data anywhere will be expensive (time and power)
  - ◆ Faults that will need to be detected and managed
    - Some detection may be the job of the programmer, as hardware detection takes power
  - ◆ Extreme scalability and performance irregularity
    - Performance will require enormous concurrency
    - Performance is likely to be variable
      - Simple, static decompositions will not scale
  - ◆ A need for latency tolerant algorithms and programming
    - Memory, processors will be 100s to 10000s of cycles away. Waiting for operations to complete will cripple performance



# Using Extra Computation in Time Dependent Problems

---

- Simple example that trades computation for a component of communication time
- Mentioned because
  - ◆ Introduces some costs
  - ◆ *Older than MPI*



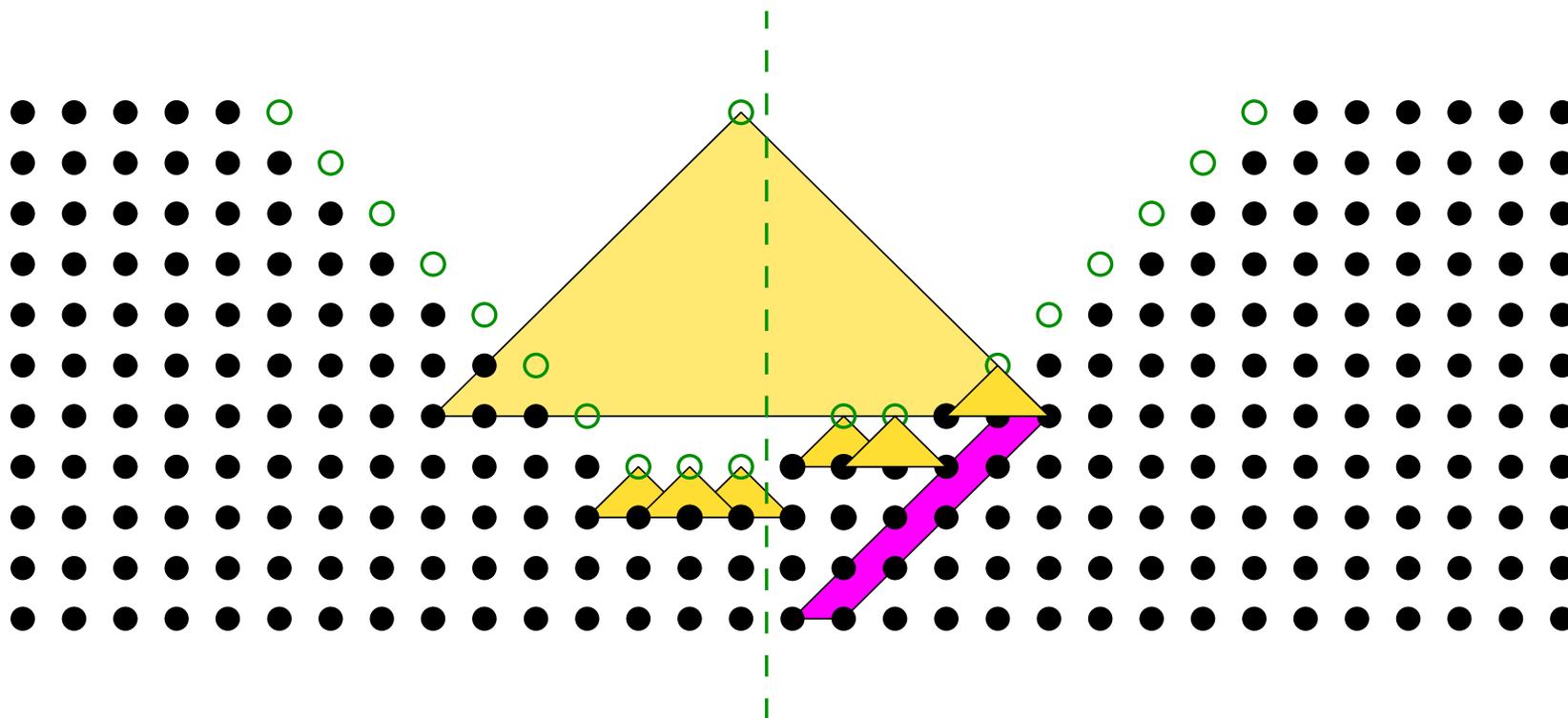
# Trading Computation for Communication

---

- In explicit methods for time-dependent PDEs, the communication of ghost points can be a significant cost
- For a simple 2-d problem, the communication time is roughly
  - ◆  $T = 4 (s + rn)$   
(using the “diagonal trick” for 9-point stencils)
  - ◆ Introduces both a communication cost and a synchronization cost (more on that later)
  - ◆ Can we do better?



# 1-D Time Stepping



# Analyzing the Cost of Redundant Computation

- Advantage of redundant computation:
  - ◆ Communication costs:
    - $K$  steps, 1 step at a time:  $2k(s+w)$
    - $K$  steps at once:  $2(s+kw)$
  - ◆ Redundant computation is roughly
    - $Ak^2c$ , for  $A$  operations for each eval and time  $c$  for each operation
- Thus, redundant computation better (under this model) when
  - ◆  $Ak^2c < 2(k-1)s$
- Since  $s$  on the order of  $10^3c$ , significant savings are possible



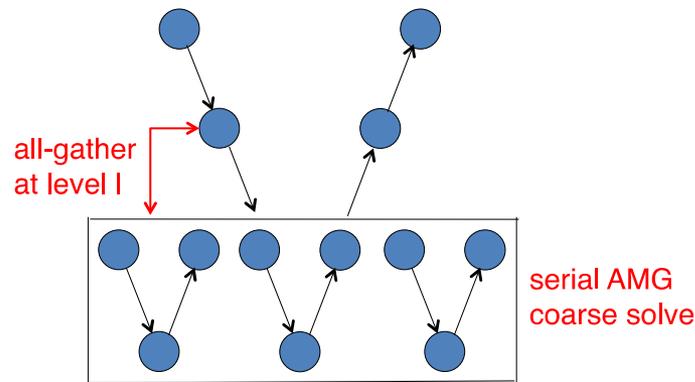
# Relationship to Implicit Methods

- A single time step, for a linear PDE, can be written as
  - ◆  $u^{k+1} = Au^k$
- Similarly,
  - ◆  $u^{k+2} = Au^{k+1} = A^2u^k = A^2u^k$
  - ◆ And so on
- Thus, this approach can be used to efficiently compute
  - ◆  $Ax, A^2x, A^3x, \dots$
- In addition, this approach can provide better temporal locality and has been developed (several times) for cache-based systems
- Why don't all applications do this?



# Using Redundant Solvers

- AMG requires a solve on the coarse grid



- Rather than either solve in parallel (too little work for the communication) or solve in serial and distribute solution, solve redundantly (either in smaller parallel groups or serial, as in this illustration)

# Redundant Solution

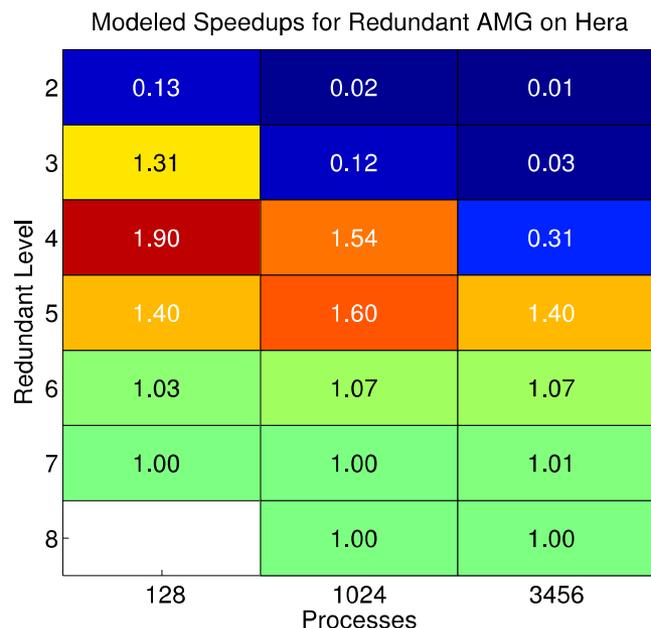
---

- Replace communication at levels  $\geq I_{red}$  with Allgather
- Every process now has complete information; no further communication needed
- Performance analysis (based on Gropp & Keyes 1989) can guide selection of  $I_{red}$



# Redundant Solves

- Applied to Hera at LLNL, provides significant speedup



- Thanks to Hormozd Gahvari



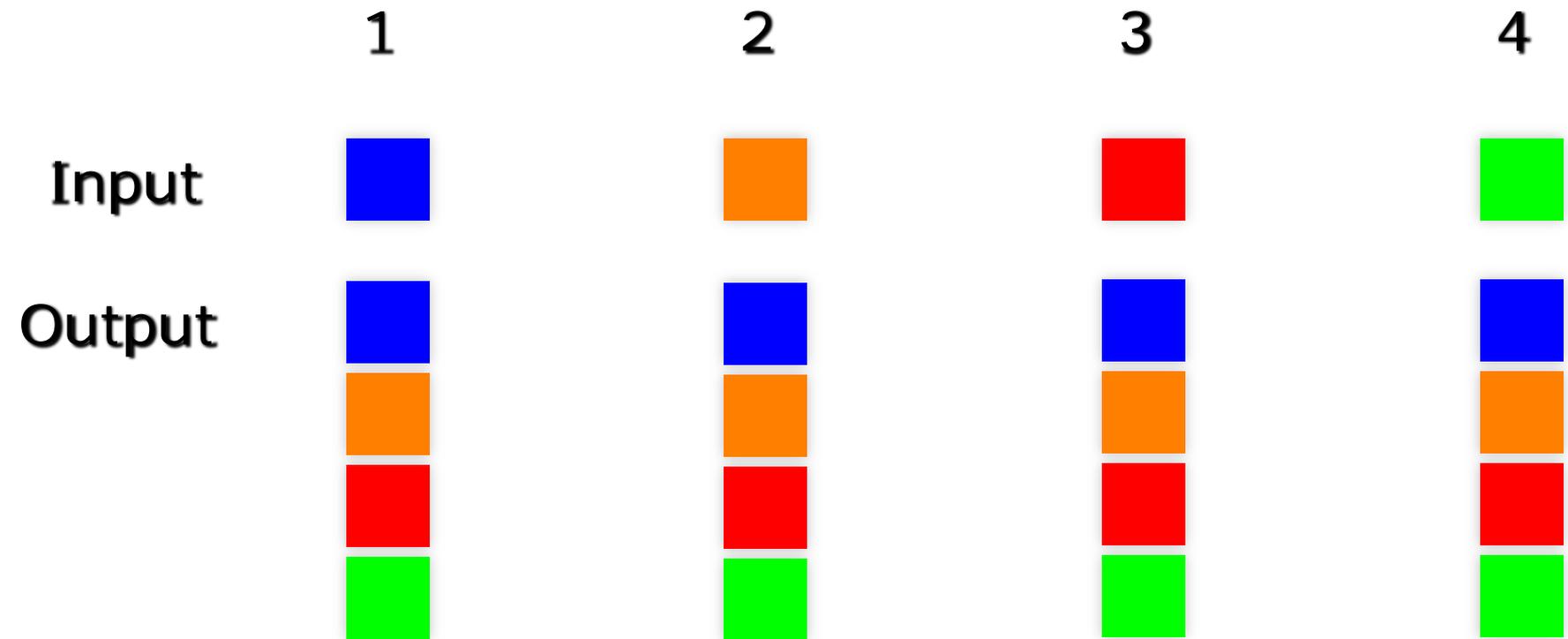
# Is it communication avoiding or minimum solution time?

---

- Example: non minimum collective algorithms
- Work of Paul Sack; see “Faster topology-aware collective algorithms through non-minimal communication”, Best Paper, PPOPP 2012
- Lesson: minimum communication need not be optimal



# Allgather



# Allgather: recursive doubling

---

a ↔ b

c ↔ d

e ↔ f

g ↔ h

i ↔ j

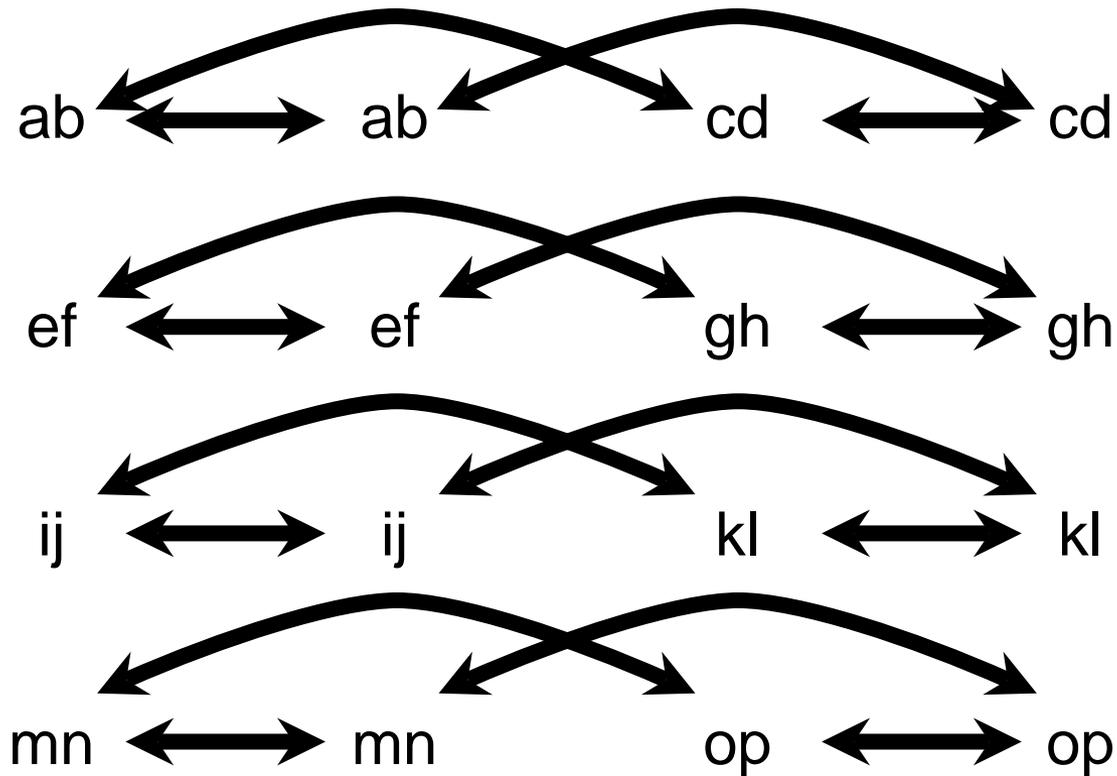
k ↔ l

m ↔ n

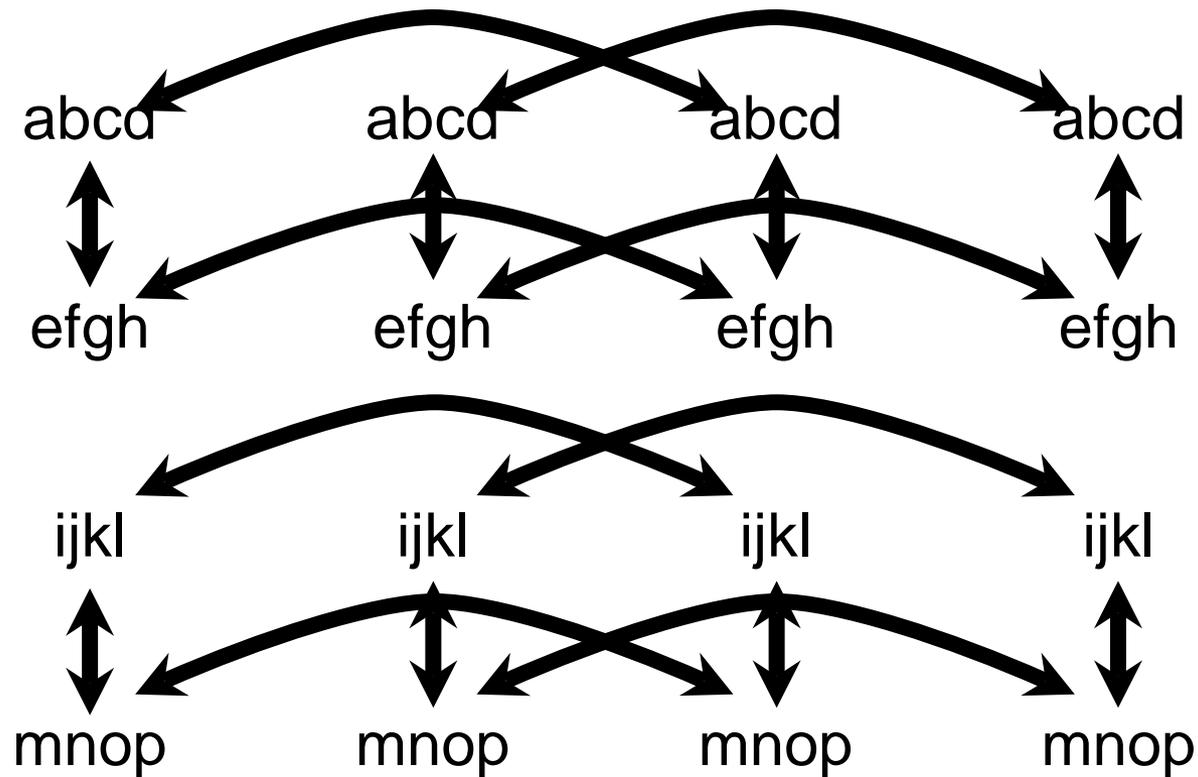
o ↔ p



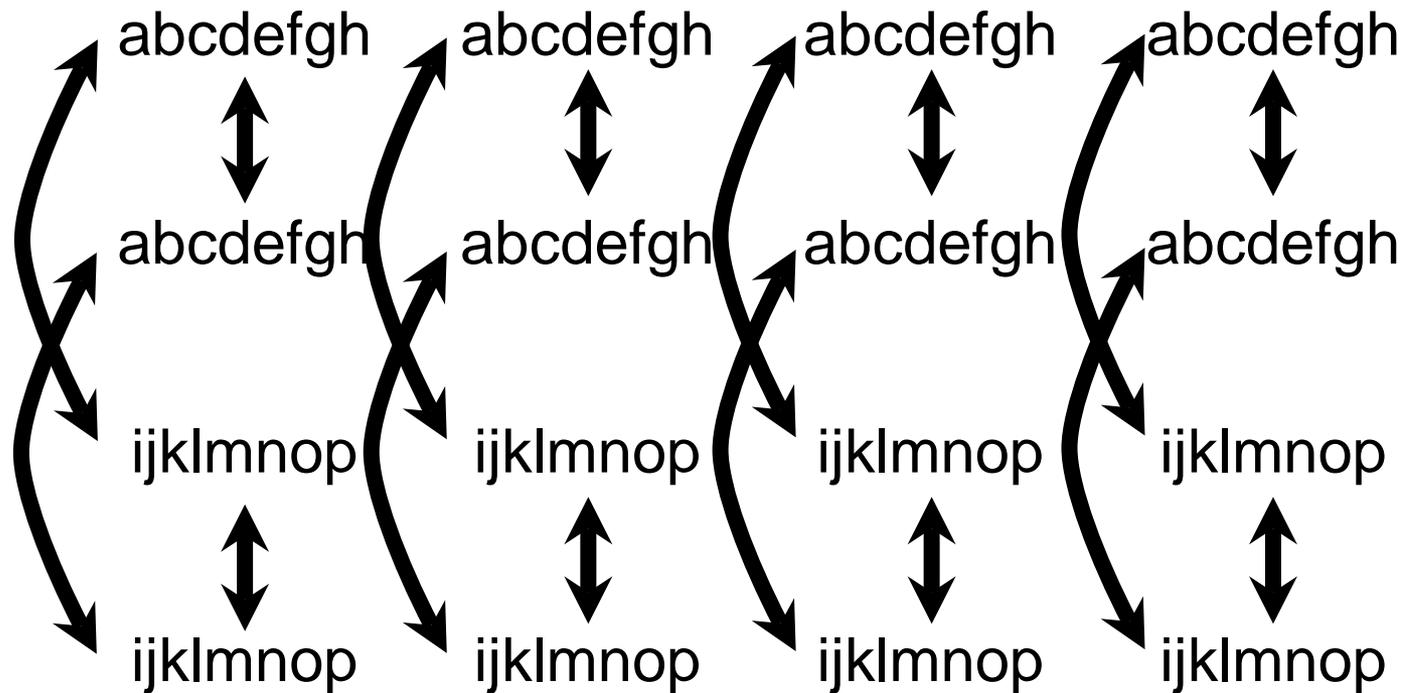
# Allgather: recursive doubling



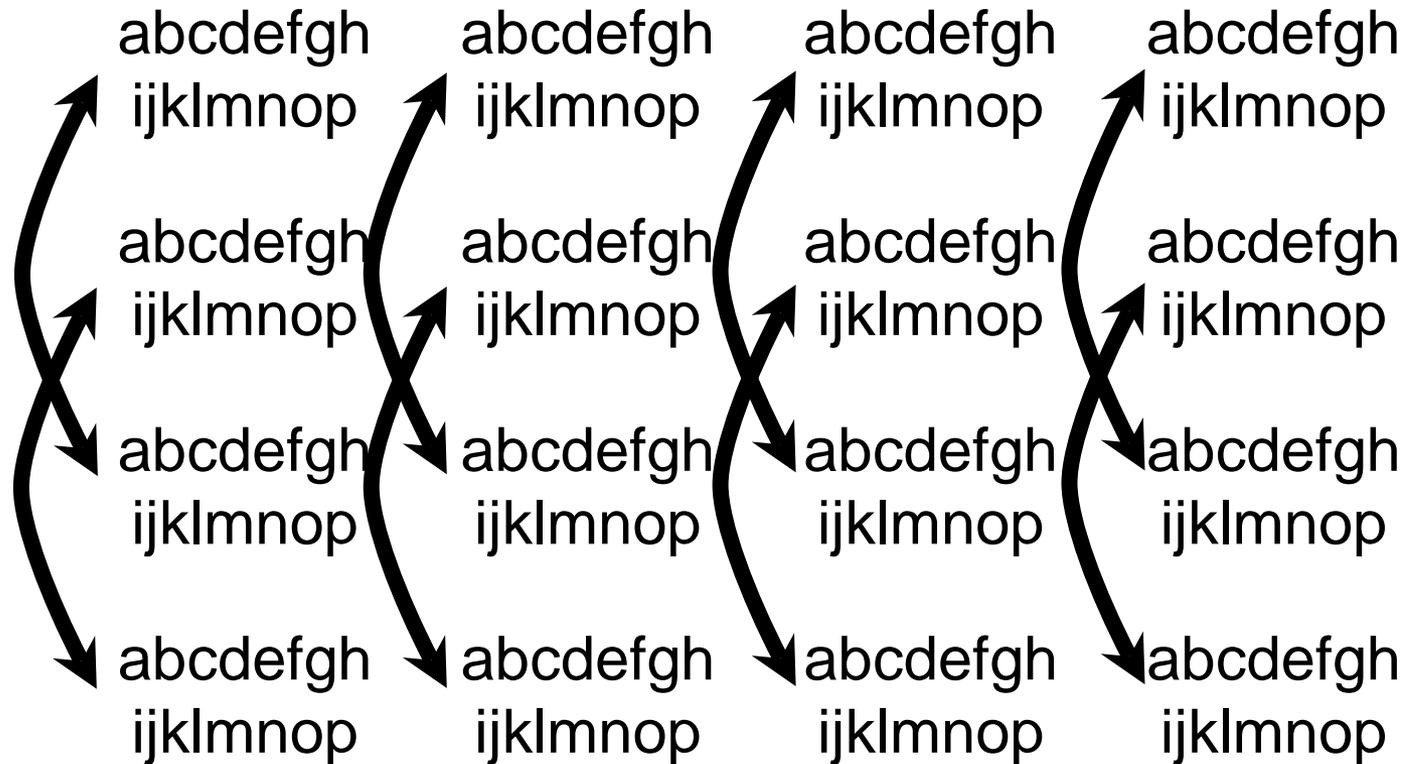
# Allgather: recursive doubling



# Allgather: recursive doubling



# Allgather: recursive doubling



$$T = (\lg P) \alpha + n(P-1)\beta$$



# Problem: Recursive-doubling

---

- No congestion model:
  - ◆  $T = (\lg P)\alpha + n(P-1)\beta$
- Congestion on torus:
  - ◆  $T \approx (\lg P)\alpha + (5/24)nP^{4/3}\beta$
- Congestion on Clos network:
  - ◆  $T \approx (\lg P)\alpha + (nP/\mu)\beta$
- Solution approach: move smallest amounts of data the longest distance



# New problem: data misordered

---

- Solution: shuffle input data
  - ◆ Could shuffle at end (redundant work; all processes shuffle)
  - ◆ Could use non-contiguous data moves
  - ◆ But best approach is often to shuffle data on network (see paper for details)



# Evaluation:

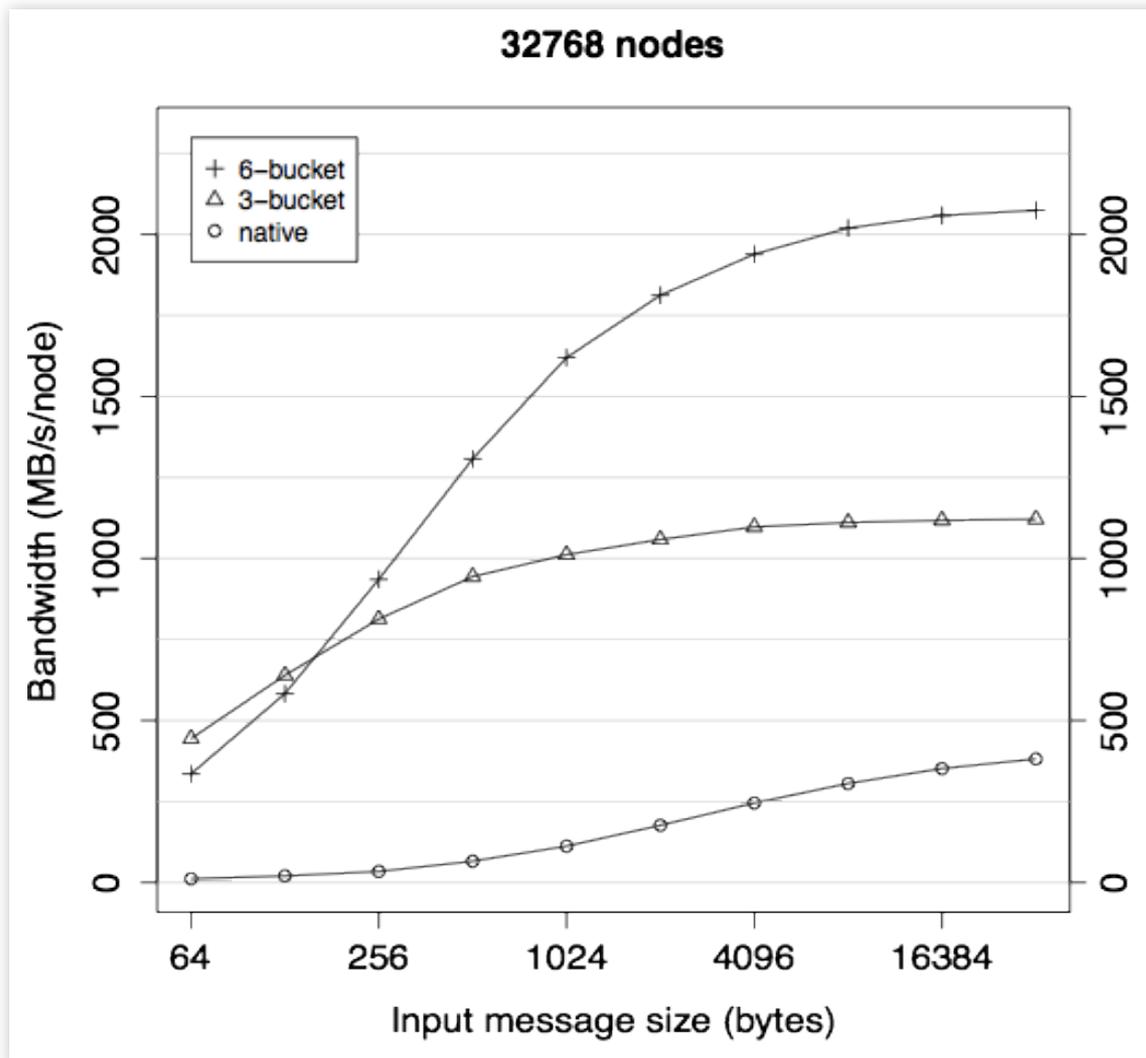
## Intrepid BlueGene/P at ANL

---

- 40k-node system
  - ◆ Each is 4 x 850 MHz PowerPC 450
- 512+ nodes is 3d torus; fewer is 3d mesh
- XLC -O4
- 375 MB/s delivered per link
  - ◆ 7% penalty using all 6 links both ways



# Allgather performance



# Notes on Allgather

---

- Bucket algorithm (not described here) exploits multiple communication engines on BG
- *Analysis shows performance near optimal*
- Alternative to reorder data step is in-memory move; analysis shows similar performance and measurements show reorder step faster on tested systems



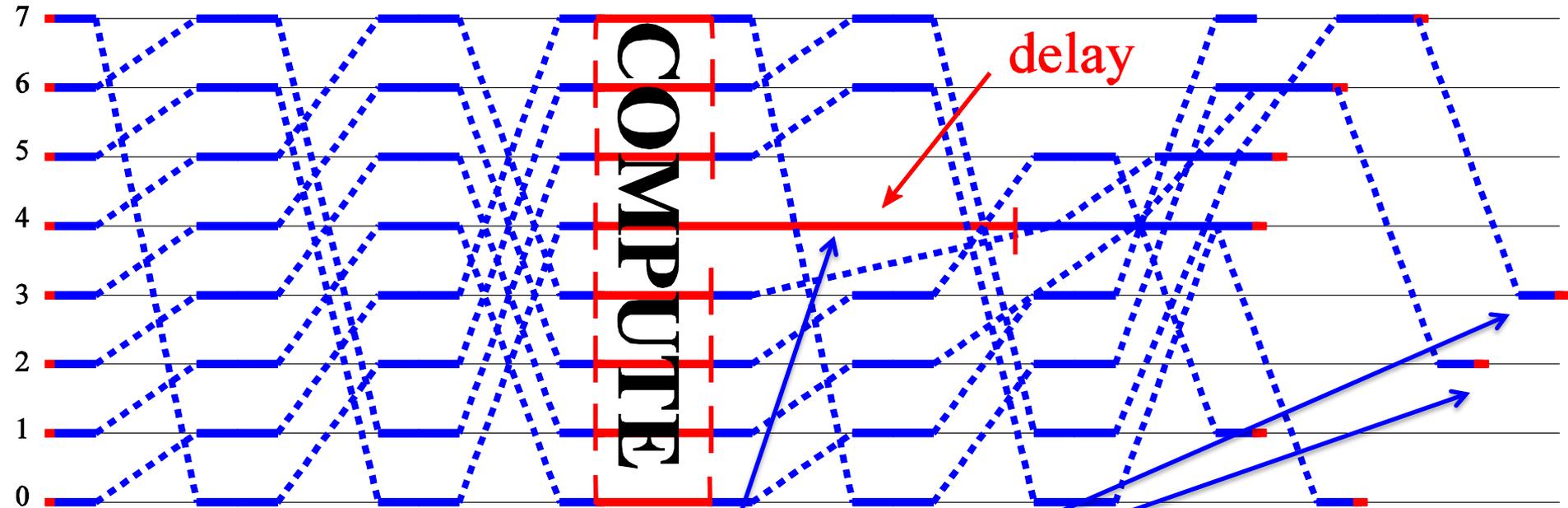
# Synchronization and OS Noise

---

- “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation,”  
Torsten Hoefler, Timo Schneider,  
Andrew Lumsdaine
  - ◆ Best Paper, SC10
- Next 3 slides based on this talk...



# A Noisy Example – Dissemination Barrier



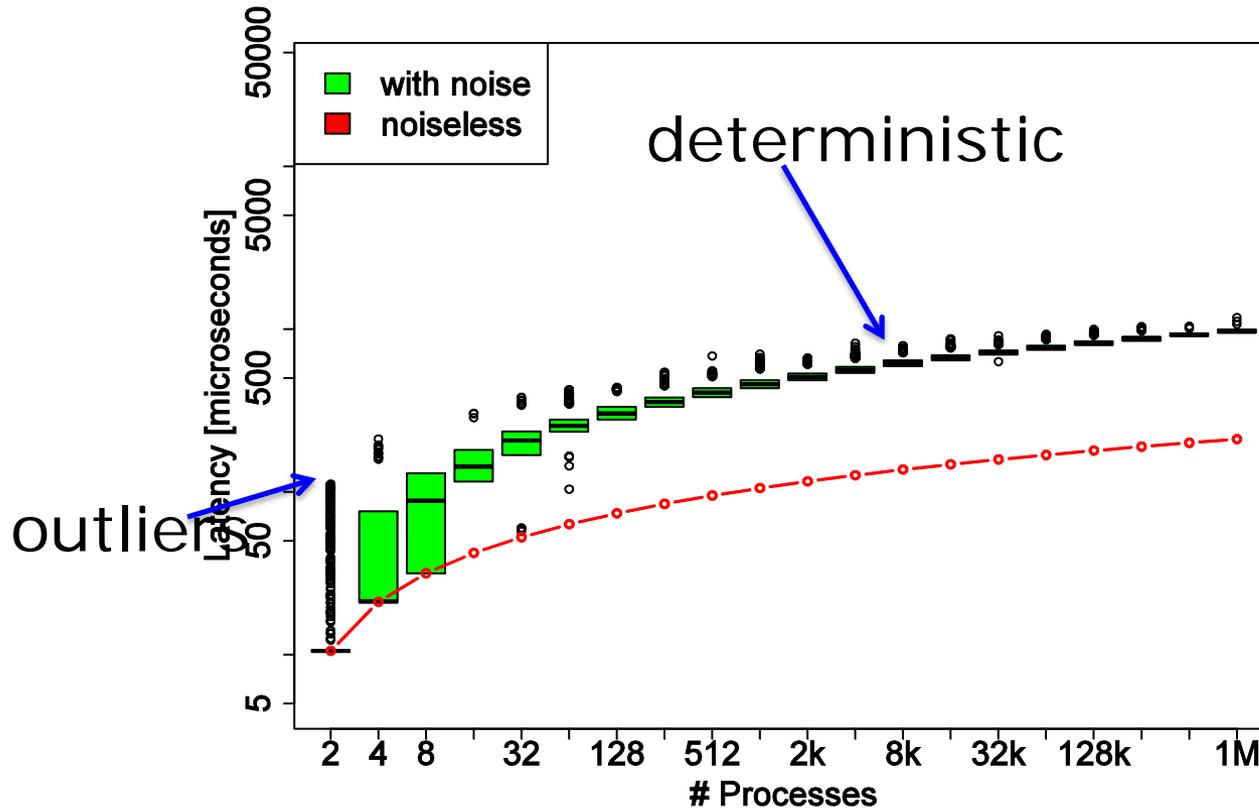
- Process 4 is delayed
  - ◆ Noise propagates "*wildly*" (of course deterministic)

# LogGOPS Simulation Framework

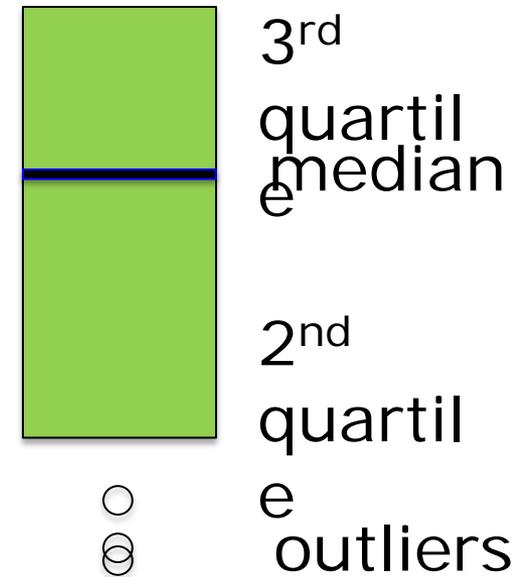
- Detailed analytical modeling is hard!
- Model-based (LogGOPS) simulator
  - ◆ Available at: <http://www.unixer.de/LogGOPSim>
  - ◆ Discrete-event simulation of MPI traces (<2% error) or collective operations (<1% error)
  - ◆ >  $10^6$  events per second
- Allows for trace-based noise injection
- Validation
  - ◆ Simulations reproduce measurements by Beckman and Ferreira well
- Details: Hoefler et al. LogGOPSim – Simulating Large-Scale Applications in the LogGOPS Model (Workshop on Large-Scale System and Application Performance, Best Paper)



# Single Collective Operations and Noise



Legend:



- 1 Byte, Dissemination, regular noise, 1000 Hz, 100  $\mu$ s



# The problem is *blocking* operations

---

- Simple, data-parallel algorithms easy to reason about but inefficient
  - ◆ True for decades, but ignored (memory)
- One solution: fully asynchronous methods
  - ◆ Very attractive, yet efficiency is low and there are good reasons for that
  - ◆ Blocking can be due to fully collective (e.g., Allreduce) or neighbor communications (halo exchange)
  - ◆ Can we save methods that involve global, synchronizing operations?



# Saving Allreduce

---

- One common suggestion is to avoid using Allreduce
  - ◆ But algorithms with dot products are among the best known
  - ◆ Can sometimes aggregate the data to reduce the number of separate Allreduce operations
  - ◆ But better is to reduce the impact of the synchronization by hiding the Allreduce behind other operations (in MPI, using `MPI_Iallreduce`)
- We can adapt CG to nonblocking Allreduce with some added floating point (but perhaps little time cost)



# The Conjugate Gradient Algorithm

- While (not converged)  
  nitters += 1;  
  s = A \* p;  
  t = p' \* s;  
  alpha = gmma / t;  
  x = x + alpha \* p;  
  r = r - alpha \* s;  
  if rnorm2 < tol2 ; break ; end  
  z = M \* r;  
  gmmaNew = r' \* z;  
  beta = gmmaNew / gmma;  
  gmma = gmmaNew;  
  p = z + beta \* p;  
end



# The Conjugate Gradient Algorithm

- While (not converged)  
nitters += 1;  
s = A \* p;  
t = p' \* s;  
alpha = gmma / t;  
x = x + alpha \* p;  
r = r - alpha \* s;  
if rnorm2 < tol2 ; break ; end  
z = M \* r;  
gmmaNew = r' \* z;  
beta = gmmaNew / gmma;  
gmma = gmmaNew;  
p = z + beta \* p;  
end



# CG Reconsidered

- By reordering operations, nonblocking dot products (MPI\_Iallreduce in MPI-3) can be overlapped with other operations
- Trades extra local work for overlapped communication
  - ◆ On a pure floating point basis, the nonblocking version requires 2 more DAXPY operations
  - ◆ A closer analysis shows that some operations can be merged
- *More work does not imply more time*



# Heterogeneity

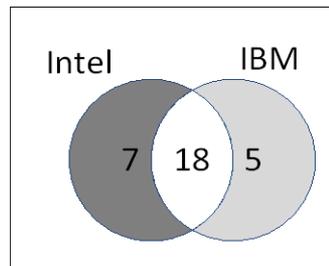
---

- Already present in complex processor architecture (e.g., “vector” operations)
- Even “identical” functional units may have different performances
- Data structure + algorithm changes
- GPUs a local hack, but some features likely to persist (different memory model, latency hiding with “threads”, vector/stream operations)

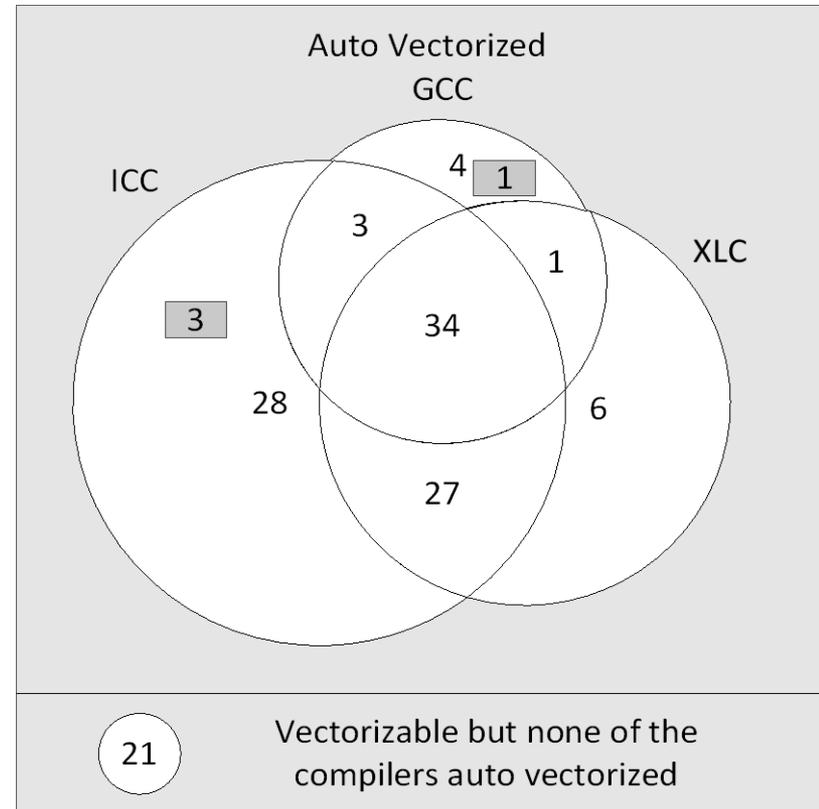


# How Good are Compilers at Vectorizing Codes?

Not Vectorizable



Vectorizable



S. Maleki, Y. Gao, T. Wong, M. Garzarán, and D. Padua. An Evaluation of Vectorizing Compilers. PACT 2011

# Processes and SMP nodes

---

- HPC users typically believe that their code “owns” all of the cores all of the time
  - ◆ The reality is that was never true, but they did have all of the cores the same fraction of time when there was one core /node
- We can use a simple performance model to check the assertion and then use measurements to identify the problem and suggest fixes.
- Based on this, we can tune a state-of-the-art LU factorization....



# Happy Medium Scheduling

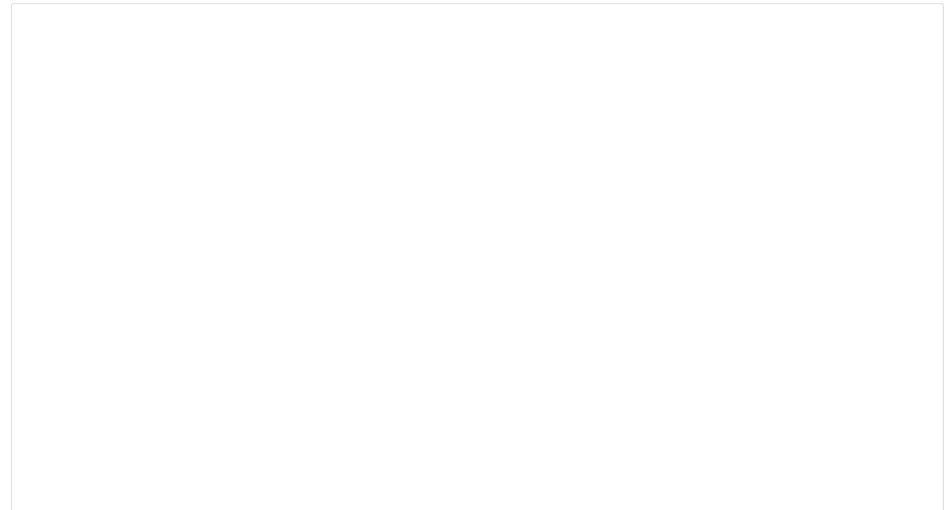
**Static** scheduling

S. Donfack, L .Grigori, V.  
Kale, WG, IPDPS '12

**Static + 10% dynamic** scheduling

**100% dynamic** scheduling

time



# Experiences

---

- Paraphrasing either Lincoln or PT Barnum:

You own some of the cores all of the time and all of the cores some of the time, but you don't own all of the cores all of the time

- Translation: a priori data decompositions that were effective on single core processors are no longer effective on multicore processors
- We see this in recommendations to “leave one core to the OS”
  - ◆ What about other users of cores, like ... the runtime system?



# What about faults?

---

- Fault *detection*
- Fault *recovery*
  - ◆ Are faults *transient* or *permanent*?
- Both can be addressed by exploiting relationships between data, implicit in algorithm
  - ◆ Simplest is conservation: gives *detection*.
  - ◆ Stokes theorem, etc. (recover from average, ECC-like data)



# Summary

---

- Extreme scale architecture *forces* us to confront architectural realities
- Even approximate (yet realistic) performance models can guide development
  - ◆ “All models are wrong; some are useful”
- Opportunities abound



# Implications (1)

---

- Restrict the use of separate computational and communication “phases”
  - ◆ Need more overlap of communication and computation to achieve latency tolerance (and energy reduction)
  - ◆ Adds pressure to be memory efficient
- May need to re-think entire solution stack
  - ◆ E.g., Nonlinear Schwarz instead of approximate Newton
  - ◆ Don't reduce everything to Linear Algebra (sorry Gene!)



# Implications (2)

---

- Use aggregates that match the hardware
- Limit scalars to limited, essential control
  - ◆ Data must be in a hierarchy of small to large
- Fully automatic fixes unlikely
  - ◆ No vendor compiles the simple code for DGEMM and uses that for benchmarks
  - ◆ No vendor compiles simple code for a shared memory barrier and uses that (e.g., in OpenMP)
  - ◆ Until they do, the best case is a human-machine interaction, with the compiler helping



# Implications (3)

---

- Use mathematics as the organizing principle
  - ◆ Continuous representations, possibly adaptive, memory-optimizing representation, lossy (within accuracy limits) but preserves essential properties (e.g., conservation)
- Manage code by using abstract-data-structure-specific languages (ADSL) to handle operations and vertical integration across components
  - ◆ So-called “domain specific languages” are really abstract-data-structure specific languages – they support more applications but fewer algorithms.
  - ◆ Difference is important because a “domain” almost certainly require flexibility with data structures and algorithms



# Implications (4)

---

- Adaptive program models with a multi-level approach
  - ◆ Lightweight, locality-optimized for fine grain
  - ◆ Within node/locality domain for medium grain
  - ◆ Regional/global for coarse grain
  - ◆ May be different programming models (hierarchies are ok!) but they must work well together
- Performance annotations to support a complex compilation environment
- Asynchronous and multilevel algorithms to match hardware



# Conclusions

---

- Planning for extreme scale systems requires rethinking both algorithms and programming approaches
- Key requirements include
  - ◆ Minimizing memory motion at all levels
  - ◆ Avoiding unnecessary synchronization at all levels
- Decisions must be informed by performance modeling / understanding
  - ◆ Not necessarily performance estimates – the goal is to guide the decisions



# Recommended Reading

---

- Bit reversal on uniprocessors (Alan Karp, SIAM Review, 1996)
- Achieving high sustained performance in an unstructured mesh CFD application (W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, B. F. Smith, Proceedings of Supercomputing, 1999)
- Experimental Analysis of Algorithms (Catherine McGeoch, Notices of the American Mathematical Society, March 2001)
- Reflections on the Memory Wall (Sally McKee, ACM Conference on Computing Frontiers, 2004)



# Thanks

- Torsten Hoefler
  - ◆ Performance modeling lead, Blue Waters; MPI datatype
- David Padua, Maria Garzaran, Saeed Maleki
  - ◆ Compiler vectorization
- Dahai Guo
  - ◆ Streamed format exploiting prefetch
- Vivek Kale
  - ◆ SMP work partitioning
- Hormozd Gahvari
  - ◆ AMG application modeling
- Marc Snir and William Kramer
  - ◆ Performance model advocates
- Abhinav Bhatele
  - ◆ Process/node mapping
- Elena Caraba
  - ◆ Nonblocking Allreduce in CG
- Van Bui
  - ◆ Performance model-based evaluation of programming models
- Ankeeth Ved
  - ◆ Model-based updates to NAS benchmarks
- Funding provided by:
  - ◆ Blue Waters project (State of Illinois and the University of Illinois)
  - ◆ Department of Energy, Office of Science
  - ◆ National Science Foundation

