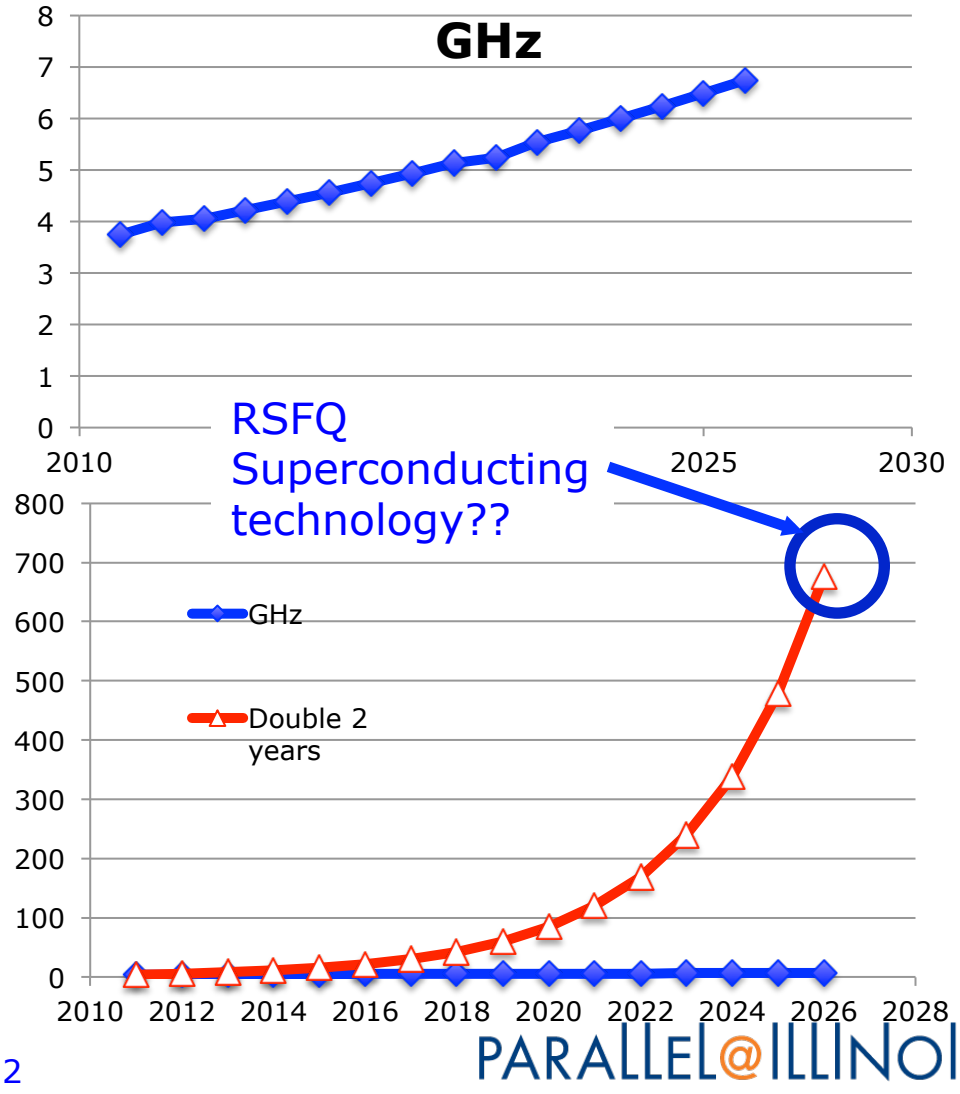
# Challenges for Algorithms and Software at Extreme Scale

William Gropp

www.cs.illinois.edu/~wgropp

# Frequency Scaling is Over

- New (prediction): Increase 4% per year (ITRS 2012 Roadmap)
- Old: Double every 2 years
- The change (loss) is enormous
- Extrapolations are just as dangerous as we tell our students

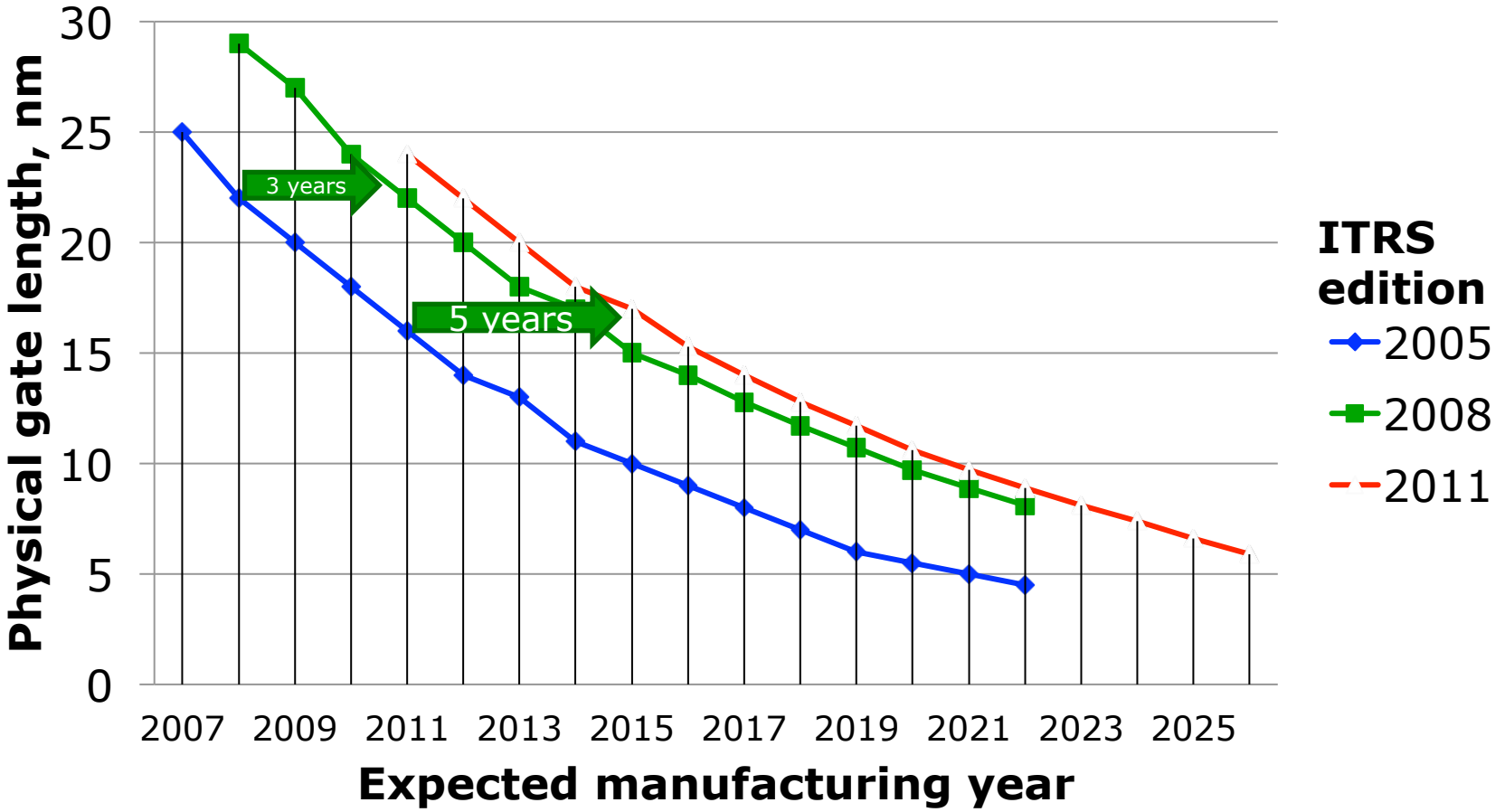






PARALLEL@ILLINOIS

# ITRS projections for gate lengths (nm) for 2005, 2008 and 2011 editions





Note the rapid 3- and then 5-year shifts in ITRS projections for physical gate lengths.



PARALLEL@ILLINOIS

# Current Petascale Systems Already Complex

- Typical processor
  - 8 floating point units, 16 integer units
    - What is a "core"?
  - Full FP performance requires use of short vector instructions
- Memory
  - Performance depends on location, access pattern
  - "Saturates" on multicore chip
- Specialized processing elements
  - E.g., NVIDIA GPU (K20X); 2688 "cores" (or 56…)
- Network
  - 3- or 5-D Torus, latency, bandwidth, contention important

# Blue Waters: NSF's Most Powerful System

- 3072 XK7 nodes and 22,752 XE6 nodes
  - ~ 1/8 GPU+CPU, 7/8 CPU+CPU
  - Peak perf: ~ 1/3 GPU+CPU, 2/3 CPU+CPU
- 1.5 PB Memory, 1TB/Sec I/O Bandwidth
- System *sustains* > **1 PetaFLOPS** on a wide range of applications
  - From starting to read input from disk to results written to disk, not just computational kernels
  - No Top500 run – does not represent application workload

# Why Is Exascale Different?

- 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 algorithms and 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

# Why is Exacale Different?

- Extreme scalability and performance irregularity
  - Performance will require enormous concurrency ($10^8$ – $10^9$)
  - 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

# What Has To Change?

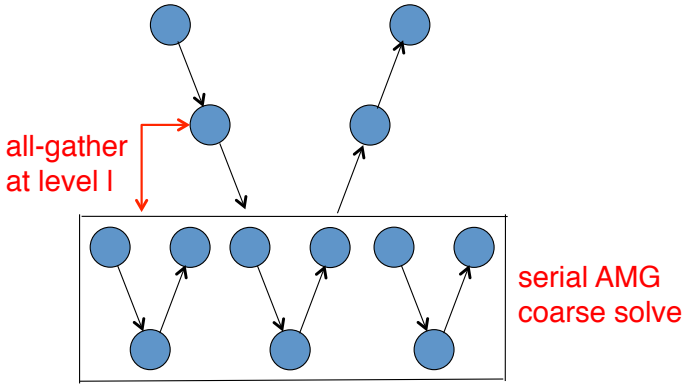
- Accept that data motion dominates cost
- Use communication cost models that include more than just point-to-point
- Provide Latency-tolerance everywhere
- Match data structure (not just algorithm) to increasingly complex hardware
- … just for starters!
- What follows are examples of how this is important *now*

# Data Motion Can Dominate Cost

- This is not new
  - Even though floating point operations are still the way computations are usually compared
- Minimizing time may require more computations
- Many examples
  - Lesson here is that simple cost models are often sufficient

# Using Redundant Solvers

- AMG requires a solve on the coarse grid



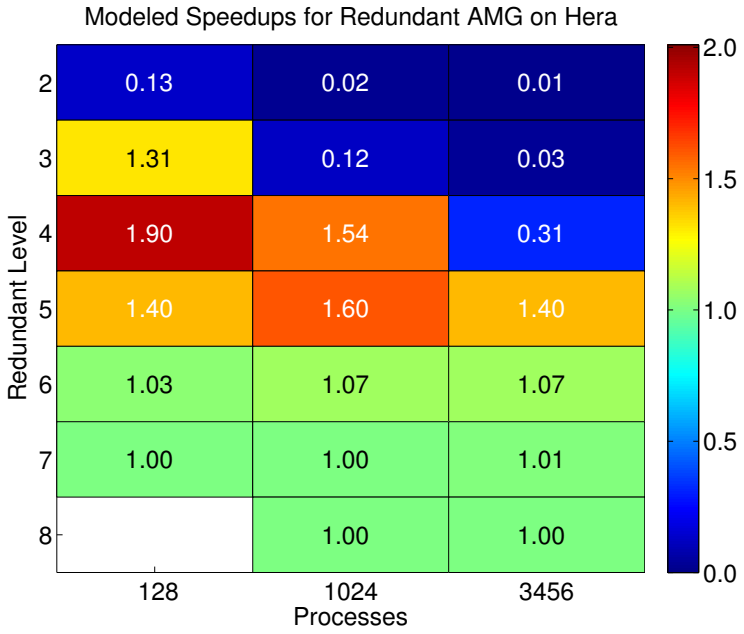

- Options:
  - Solve in parallel (too little work)
  - Solve in serial and distribute (Amdahl bottleneck + communication)
  - Solve redundantly

# Redundant Solution

- Replace communication at levels ≥L with Allgather
- Every process now has complete information; no further communication needed
  - ♦ Solution is computed redundantly
- Performance analysis (based on Gropp & Keyes 1989) can guide selection of L
  - ♦ Must be modified by characteristics of modern CPUs and networks

# Redundant Solves

- Applied to Hera at LLNL, provides significant speedup



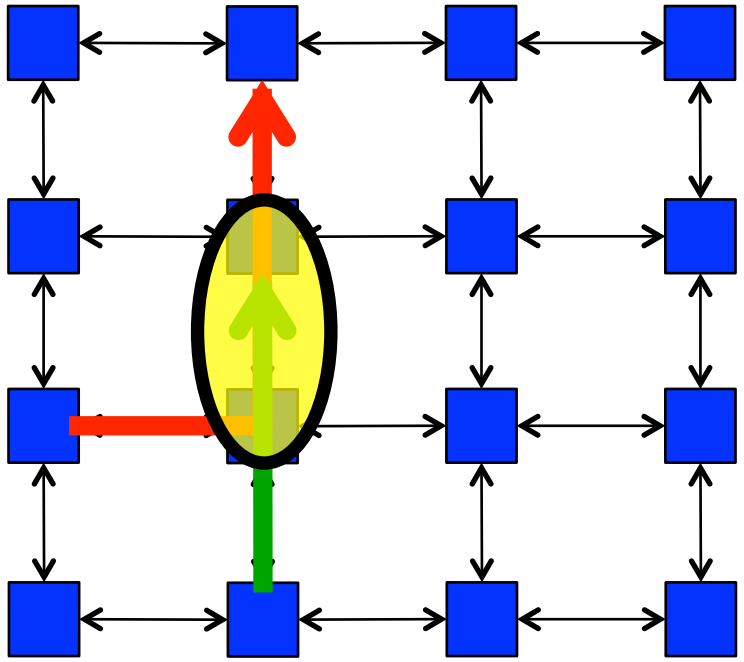

- Lesson: More work can be *faster*
- Key idea is to compute performance *envelope*
- Thanks to Hormozd Gahvari



PARALLEL@ILLINOIS

# Communication Cost Includes More than Latency and Bandwidth

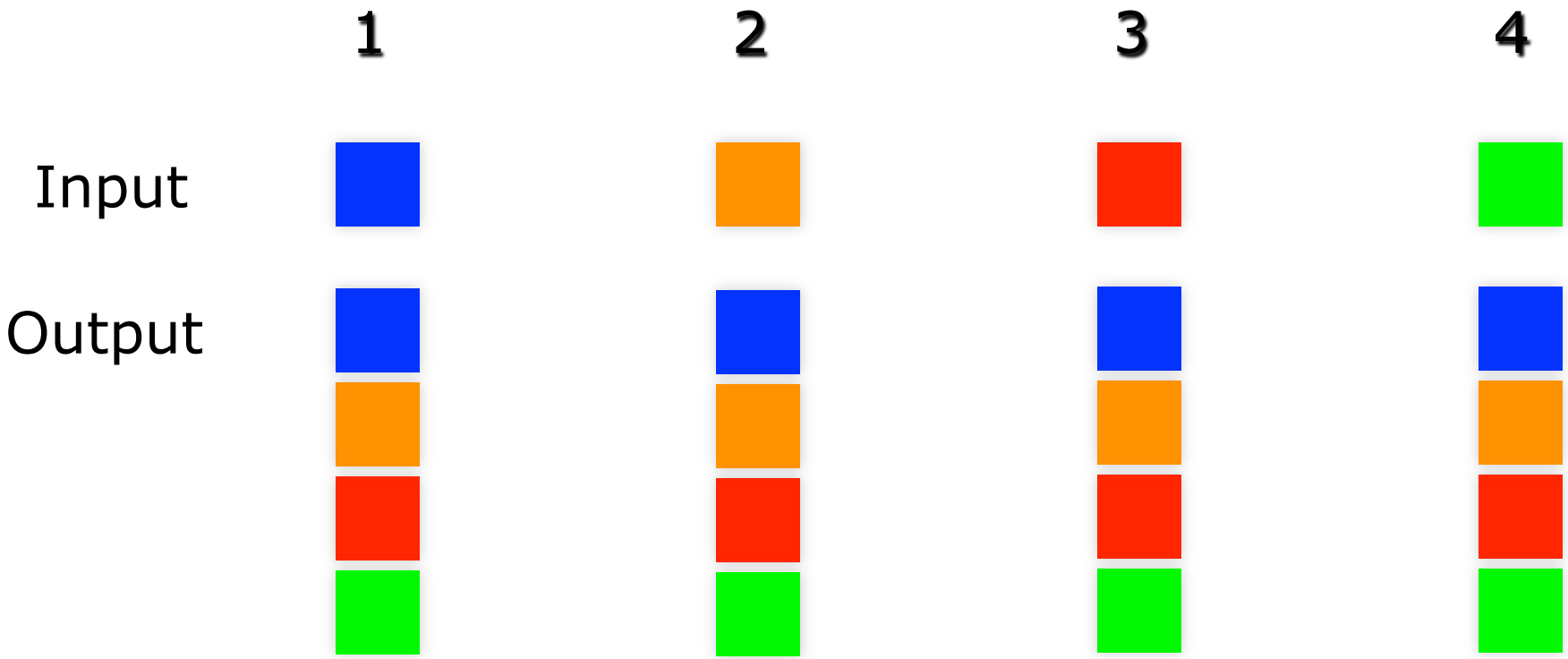
- Communication does not happen in isolation
- Effective bandwidth on shared link is ½ point-to-point bandwidth
- Real patterns can involve many more (integer factors)
- Loosely synchronous algorithms ensure communication cost is worst case



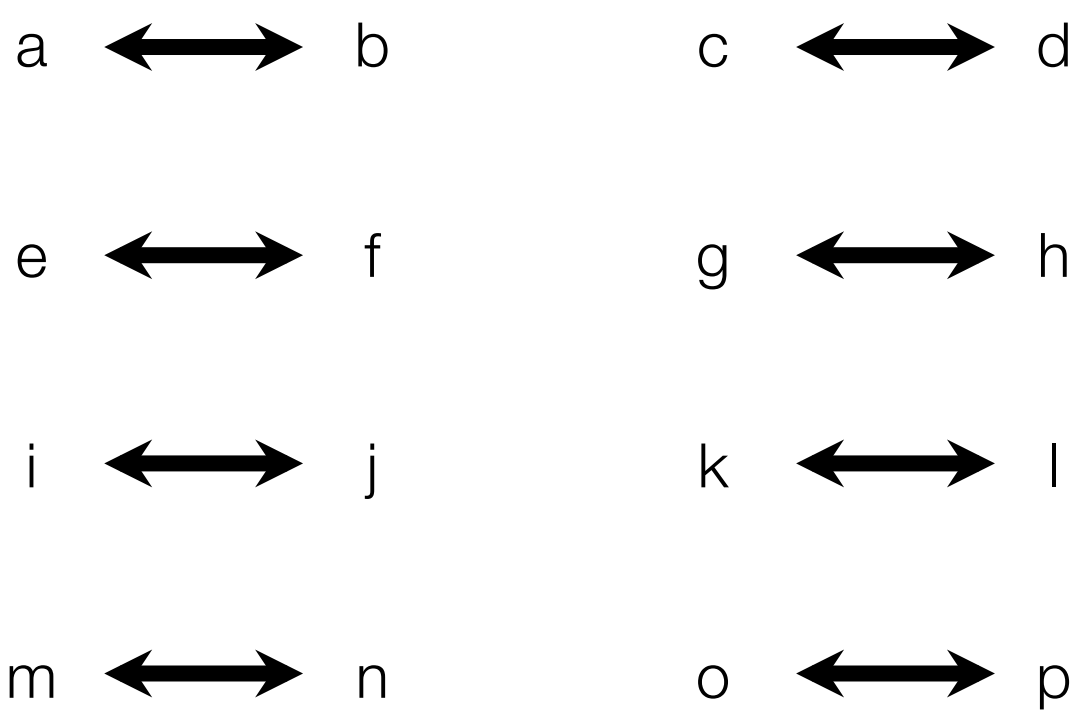
# Is It Communication Avoiding Or Minimum Solution Time?

- 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*

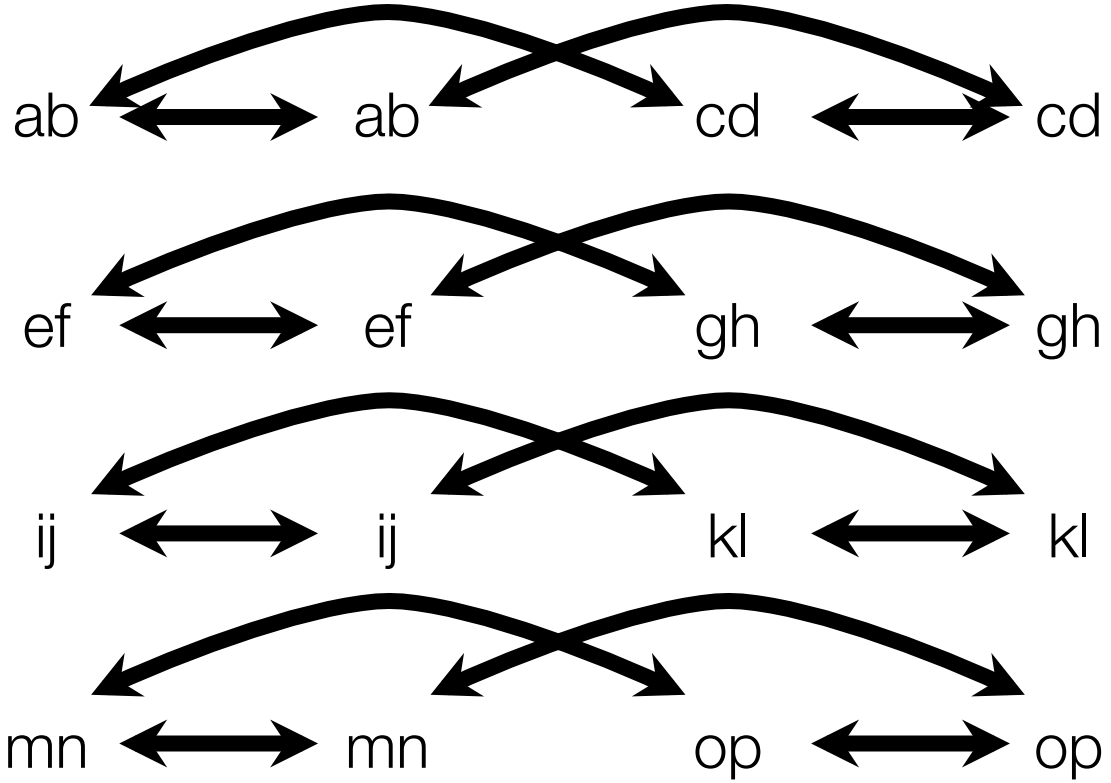
# Allgather



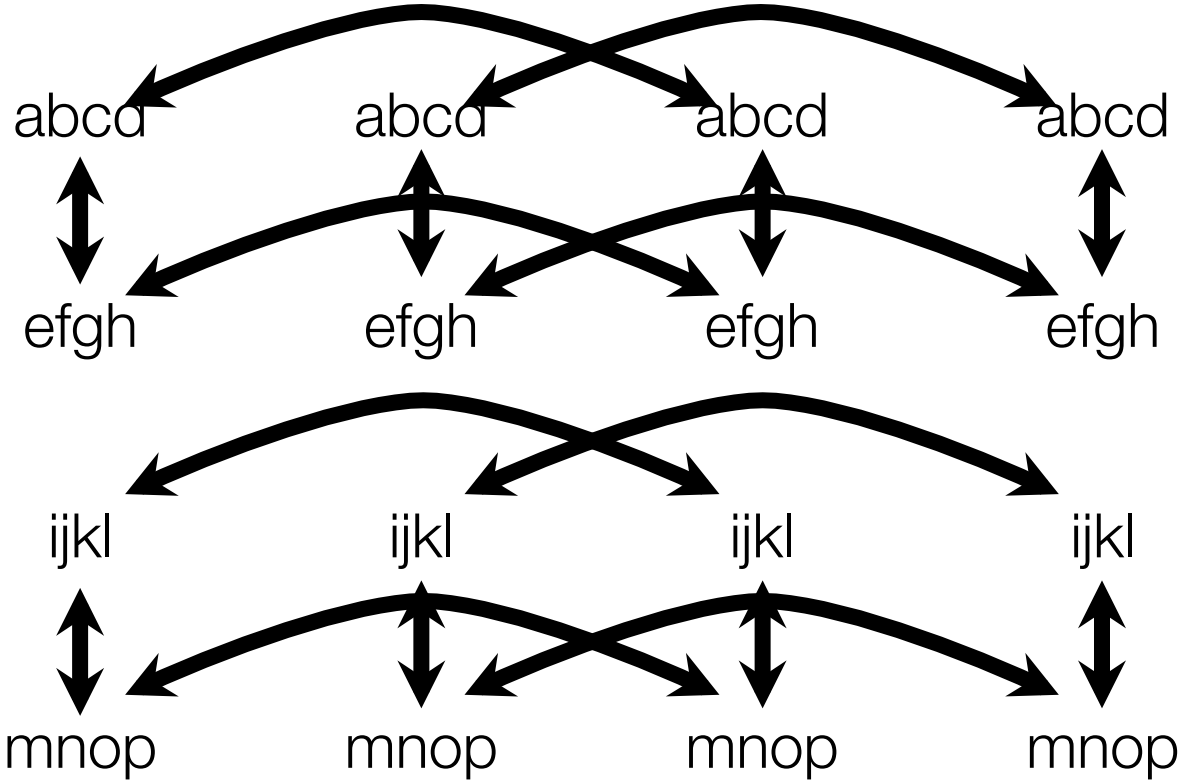

# Allgather: Recursive Doubling



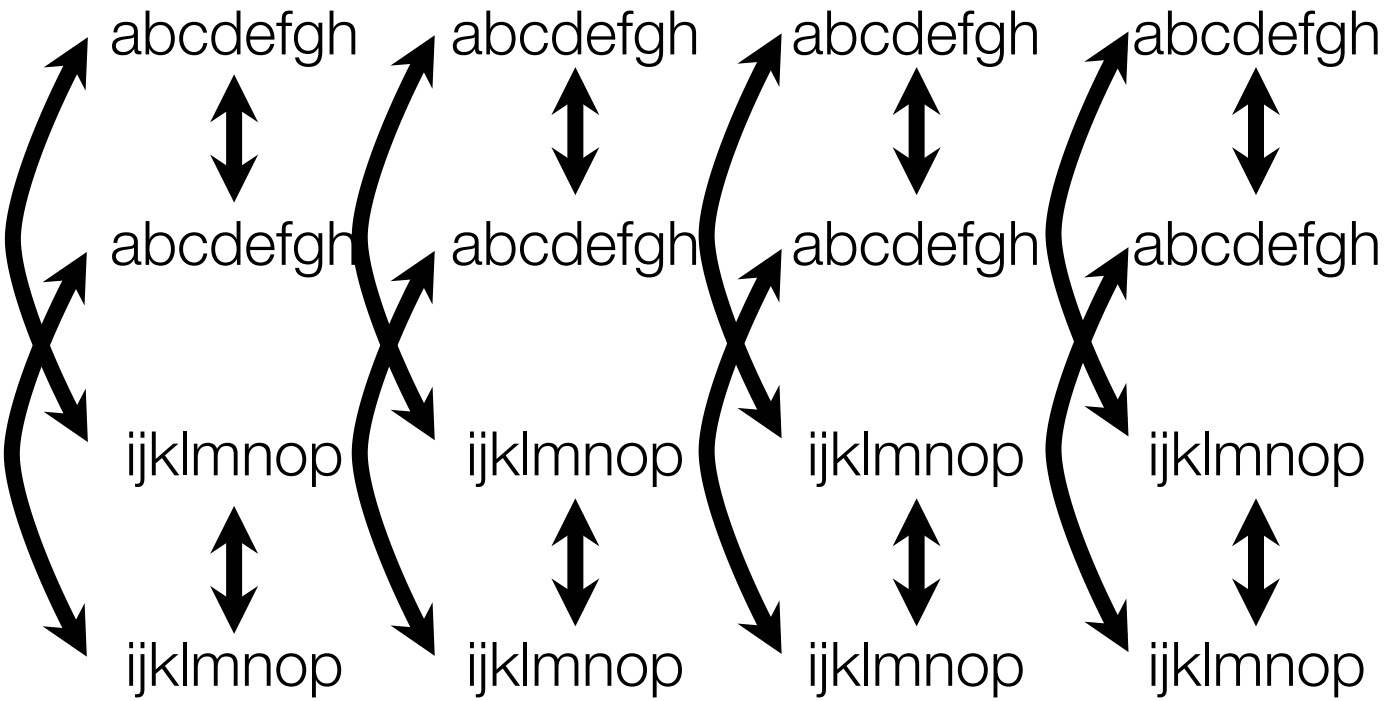

# Allgather: Recursive Doubling



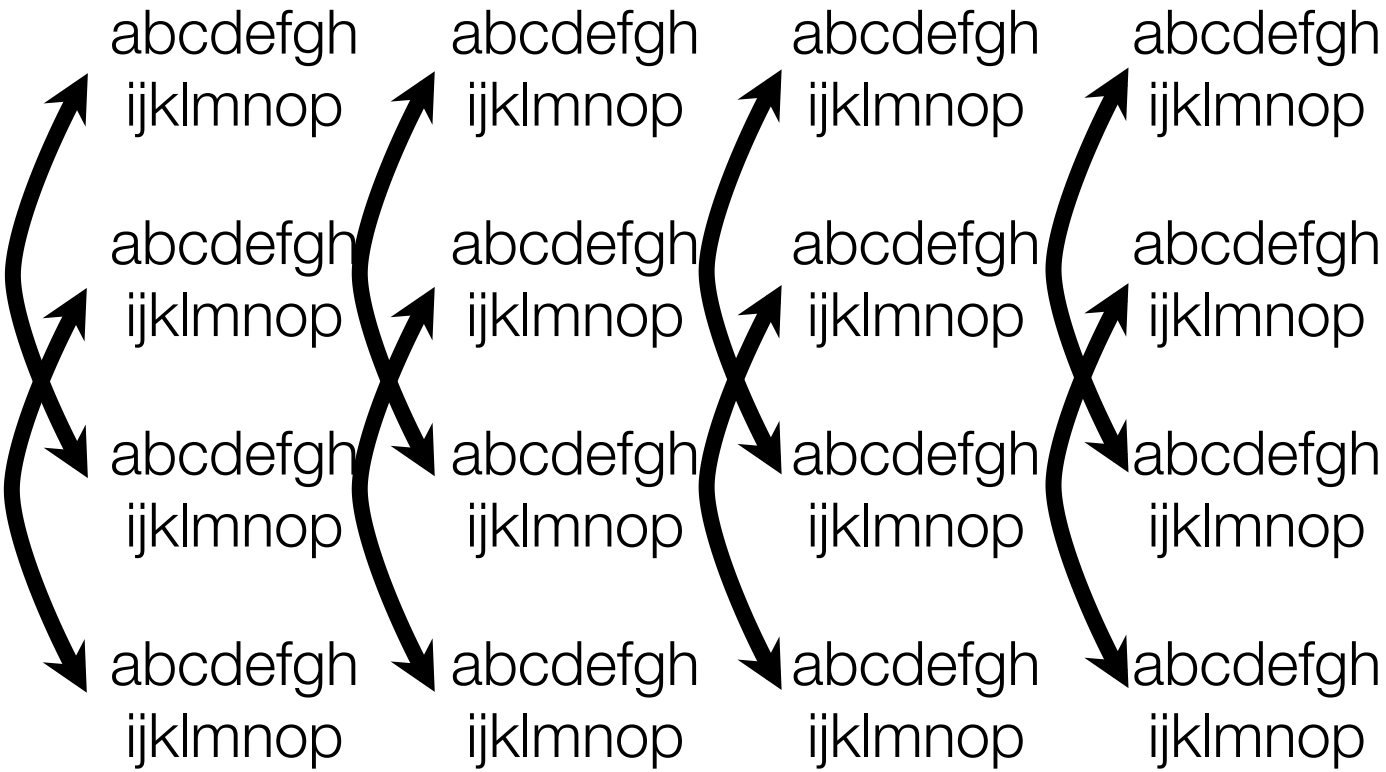

# Allgather: Recursive Doubling




# Allgather: Recursive Doubling




# Allgather: Recursive Doubling



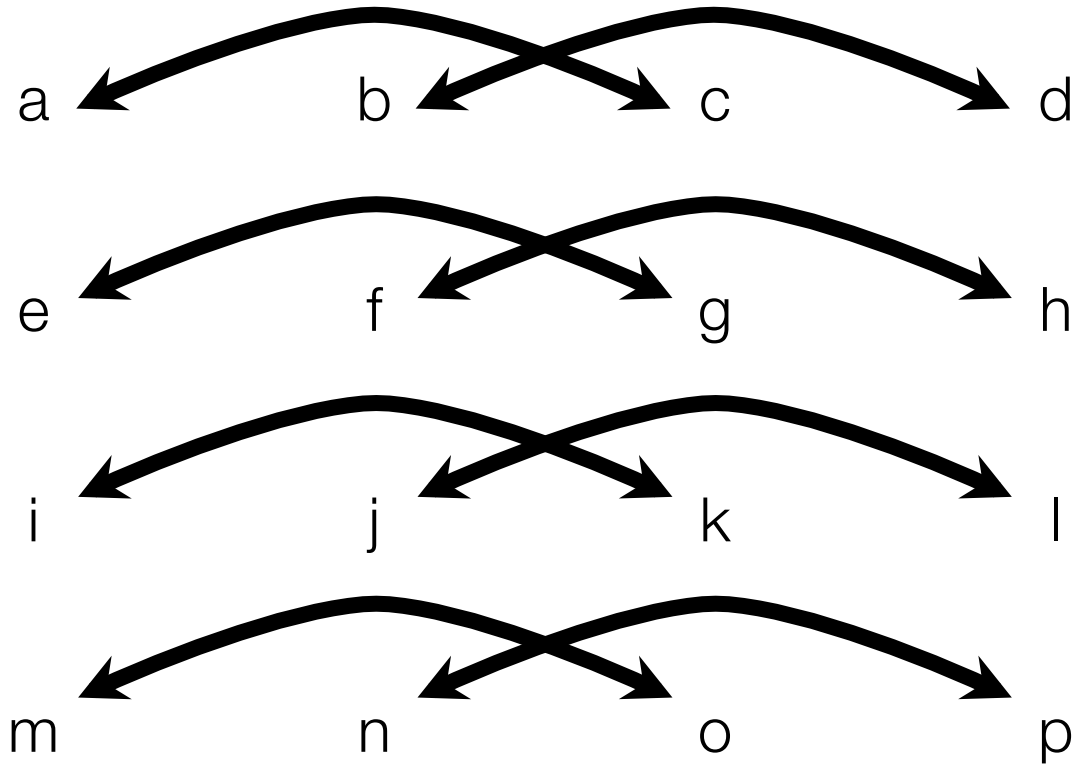

$T=(lg\ P)\ \alpha + n(P-1)\beta$

# Problem: Recursive-Doubling

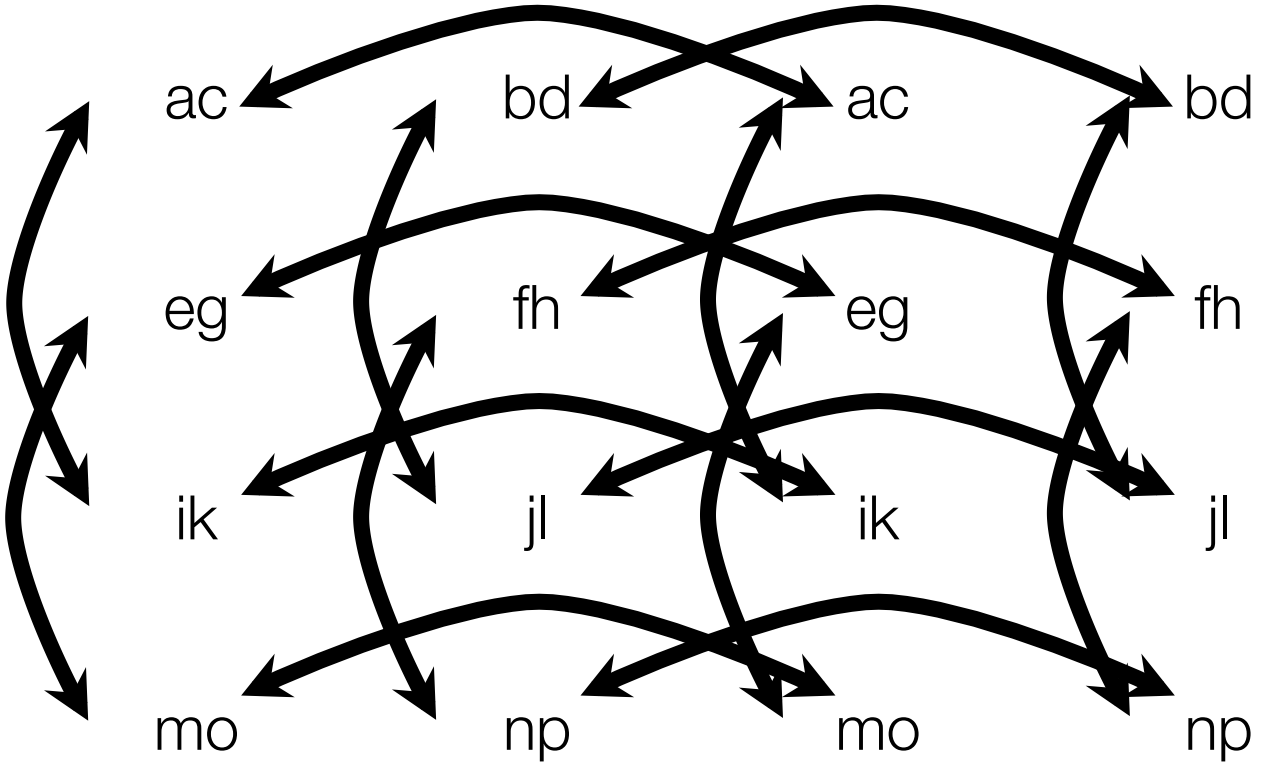
- No congestion model:
  - $T=(\lg P)\alpha + n(P-1)\beta$
- Congestion on torus:
  - $T\approx(\lg P)\alpha + (5/24)nP^{4/3}\beta$
- Congestion on Clos network:
  - $T\approx(\lg P)\alpha + (nP/\mu)\beta$

- Solution approach: move smallest amounts of data the longest distance

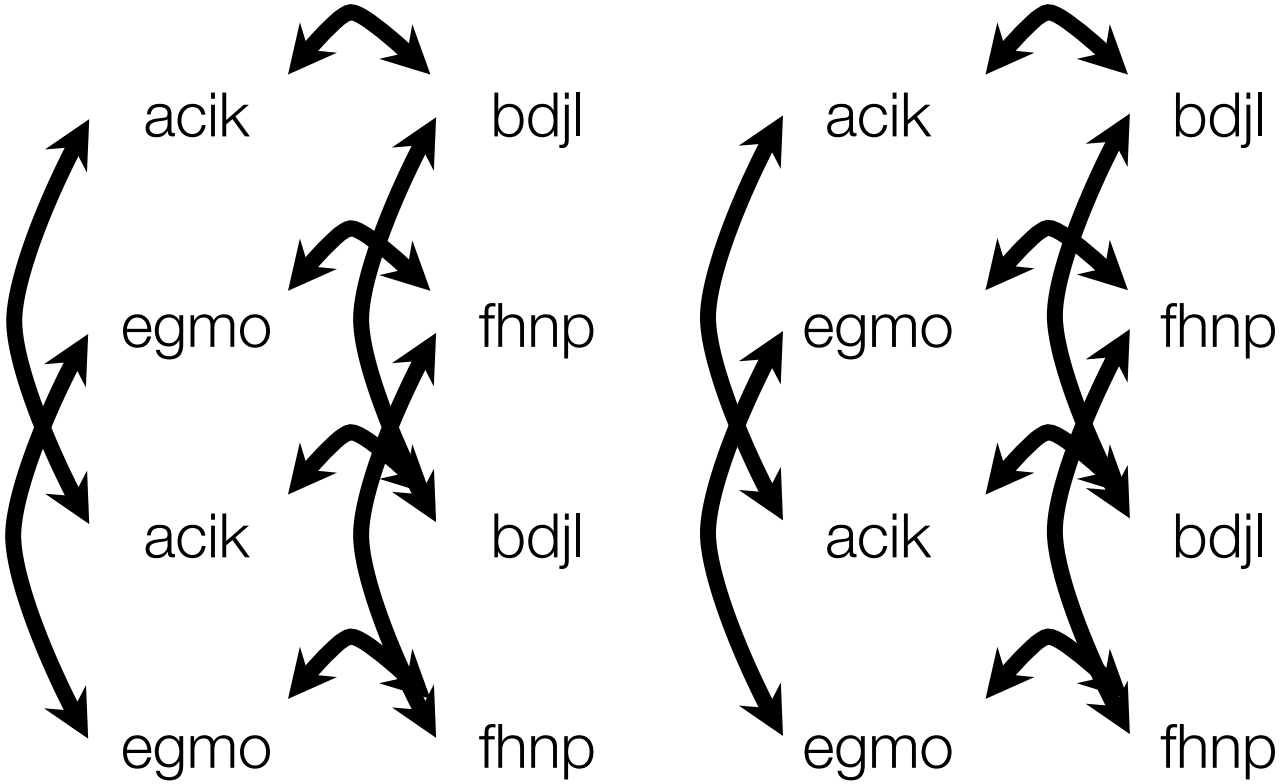
# Solution: Recursive-Doubling



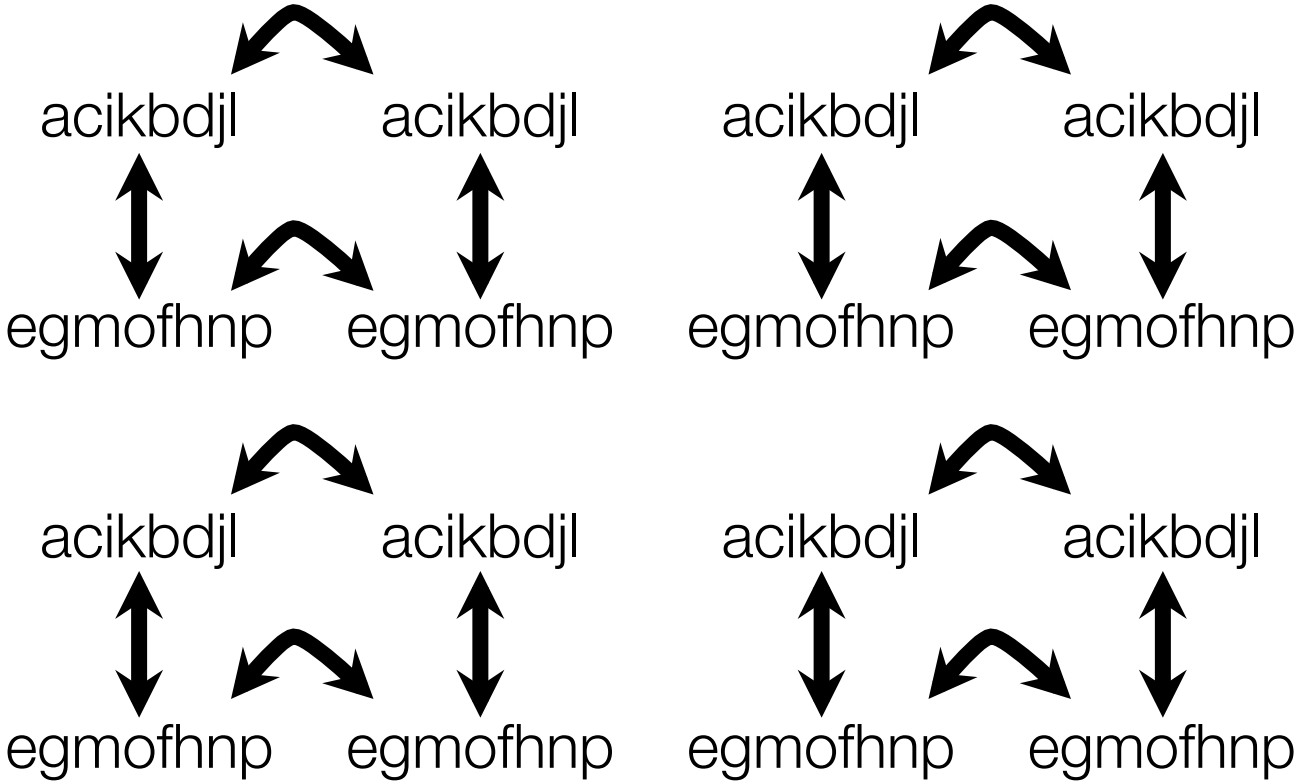

# Solution: Recursive-Doubling




# Solution: Recursive-Doubling




# Solution: Recursive-Doubling

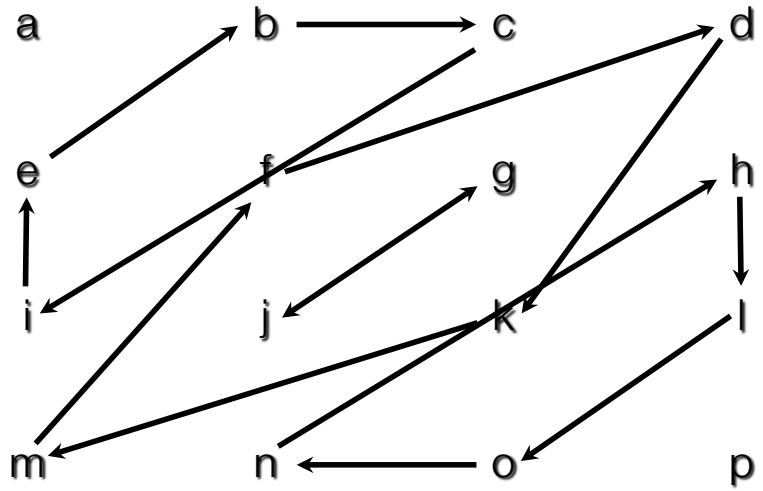



# Solution: Recursive-Doubling


$$T=(lg\ P)\alpha + (7/6)nP\beta$$

# New Problem: Data Misordered

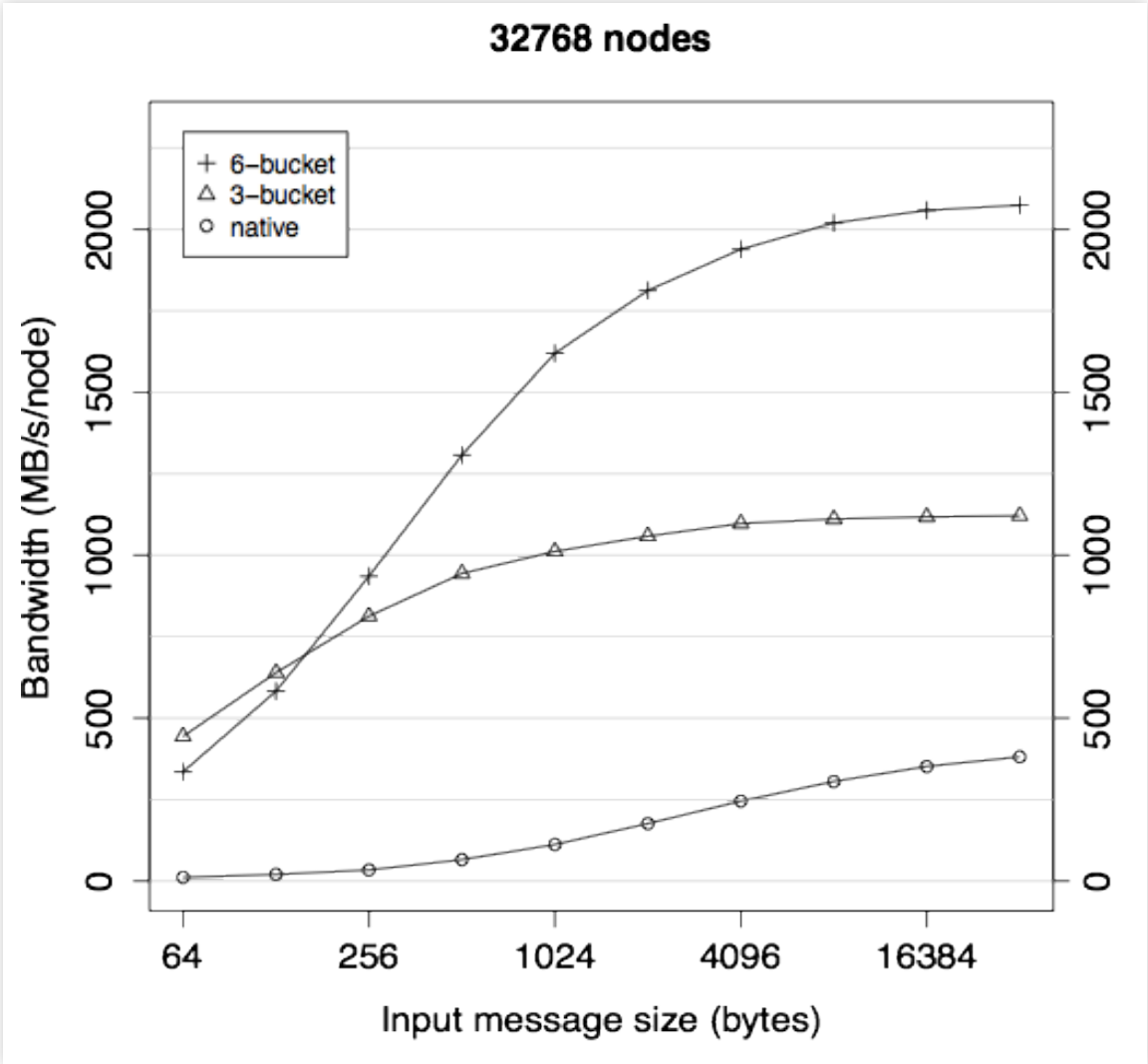
- Solution: shuffle input data
  - Could shuffle at end (redundant work; all processes shuffle)
  - Could use non-contiguous data moves (but extra overhead)
  - But best approach is often to shuffle data on network (see paper for details)




# 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






PARALLEL@ILLINOIS

# 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

# Latency Tolerance Everywhere

- Communication, even to local memory, takes 10s to 100s of cycles
  - 1000 to 10,000s in big machines to remote nodes
- Time waiting is *lost*
- Needs algorithmic help
  - Many algorithms have dependencies that are latency **in**tolerant

# Scaling Problems

- Simple, data-parallel algorithms easy to reason about but inefficient
  - True for decades, but ignored (memory)
  - Log p terms can dominate at $p = 10^6$
- One solution: fully asynchronous methods
  - Very attractive (parallel efficiency high), yet solution efficiency is low and there are good reasons for that
  - Blocking (synchronizing) communication 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-3, 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)

```
  niters += 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)

```
  niters += 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
```

# A Nonblocking Version of CG

- While (not converged)

```
 niters += 1;
 s     = Z + beta * s;
 Begin p'*s
 S     = M * s;
 Complete t      = p' *s;
 alpha = gmma / t;
 x     = x + alpha * p;
 r     = r - alpha * s;
 if rnorm2 < tol2 ; break ; end
 z     = z - alpha * S;
 Begin r'*z here (also begin r'*r for convergence test)
 Z     = A * z;
 Complete gmmaNew = r' * z;
 beta  = gmmaNew / gmma;
 gmma  = gmmaNew;
 p     = z + beta * p;
end
```

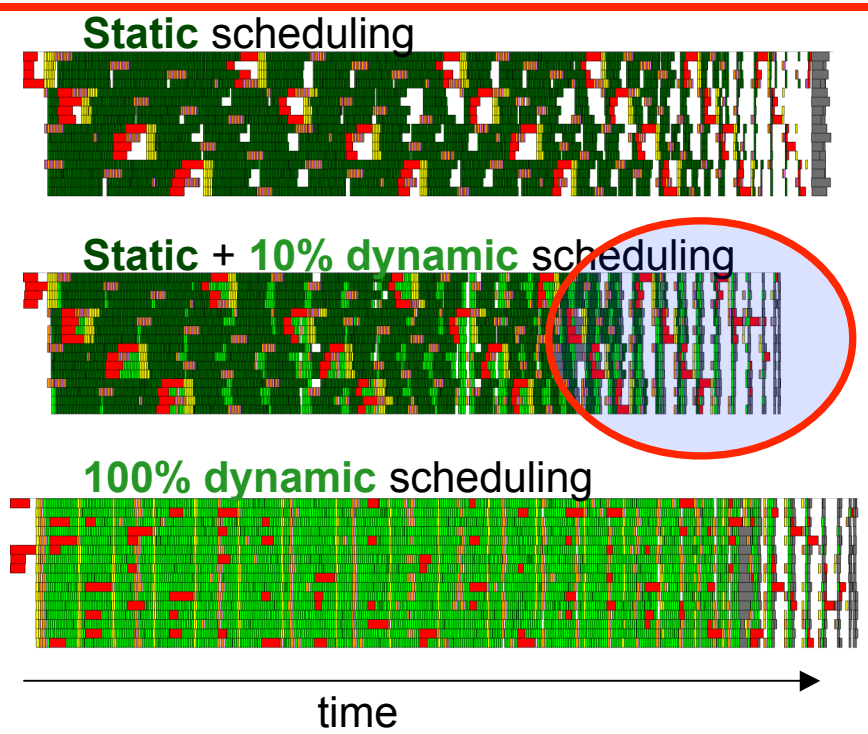
# CG Reconsidered

- 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 (in terms of memory references)
    - Count *memory motion*, not *floating point*
- Other approaches possible; see "Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm," P. Ghysels and W. Vanroose, submitted
- ***More work does not imply more time***

# 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

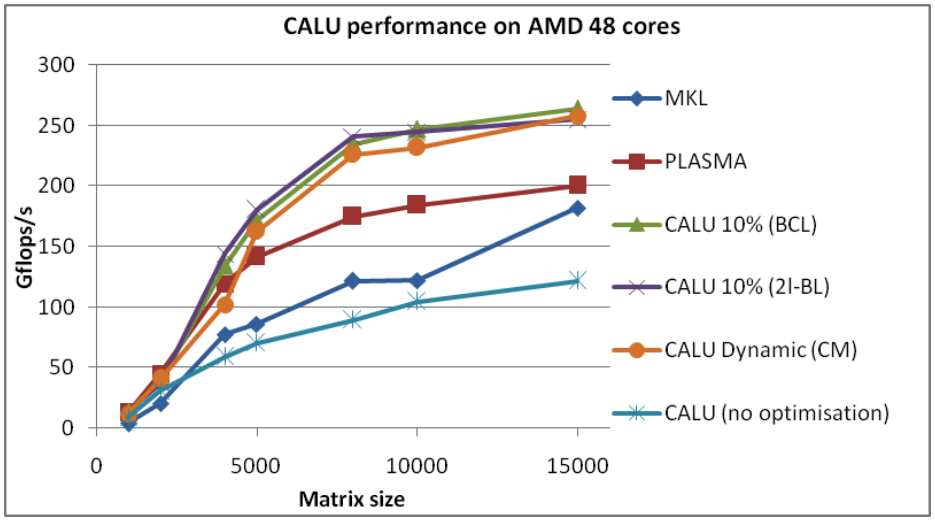



Scary Consequence: Static data decompositions *will not work at scale.*

Corollary: programming models with static task models *will not work at scale*

Performance irregularities introduce load-imbalance.
Pure dynamic has significant overhead; pure static too much imbalance.
Solution: combined static and dynamic scheduling

Communication Avoiding LU factorization (CALU) algorithm, S. Donfack, L .Grigori, V. Kale, WG, IPDPS '12






PARALLEL@ILLINOIS

# Changing Requirements for Data Decomposition

- 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?

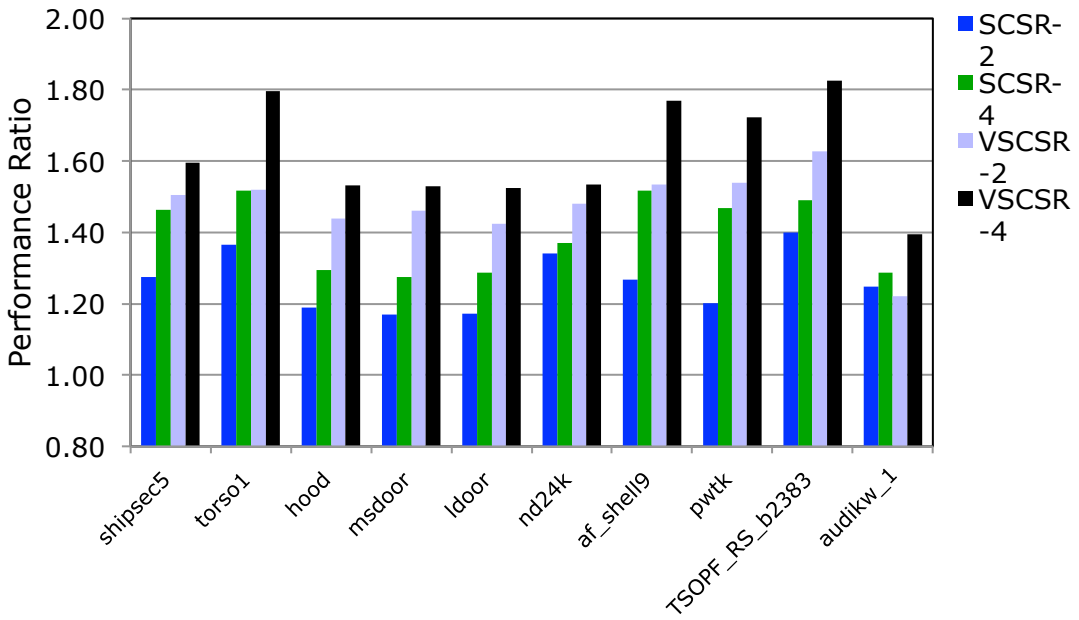
# Match Data Structure to Hardware

- Processors include hardware to address performance challenges
  - "Vector" function units
  - Memory latency hiding/prefetch
  - Atomic update features for shared memory
  - Etc.
- Both algorithms and data structures must be designed to work well with real hardware

# Sparse Matrix-Vector Multiply

**Barriers to faster code**

- "Standard" formats such as CSR do not meet requirements for prefetch or vectorization
- Modest changes to data structure enable both vectorization, prefetch, for 20-80% improvement on P7




Prefetch results in *Optimizing Sparse Data Structures for Matrix Vector Multiply*
http://hpc.sagepub.com/content/25/1/115



PARALLEL@ILLINOIS

# What Does This Mean For You?

- It is time to rethink data structures and algorithms to match the realities of memory architecture
  - For SpMV, we have results for x86 where the benefit is smaller but still significant
  - Even more important for GPUs
  - Better match of algorithms to prefetch hardware is necessary to overcome memory performance barriers
- Similar issues come up with heterogeneous processing elements (someone needs to *design* for memory motion and concurrent and nonblocking data motion)

# 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 requires 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, MPI datatype
- 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
- Van Bui
  - Performance model-based evaluation of programming models
- Collaborators including
  - Kirk Jordan, Martin Schultz, Ulrike Yang, Todd Gablin, Bronis de Supinski, …
- Funding provided by:
  - Blue Waters project (State of Illinois and the University of Illinois)
  - Department of Energy, Office of Science
  - National Science Foundation