# Finding the Happy Medium: Tradeoffs in Communication, Algorithms, Architectures and Programming Models

William Gropp

www.cs.illinois.edu/~wgropp

# Why is this the "Architecture" Talk?

- Algorithms and software must acknowledge realities of architecture
- Can encourage architecture changes
  - ♦ But must have compelling case – cost of trying is high, even with simulation
- Message:
  - ♦ ***Appropriate*** performance models can guide development
  - ♦ Avoid ***unnecessary*** synchronization
    - Often encouraged by the programming model

# 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)

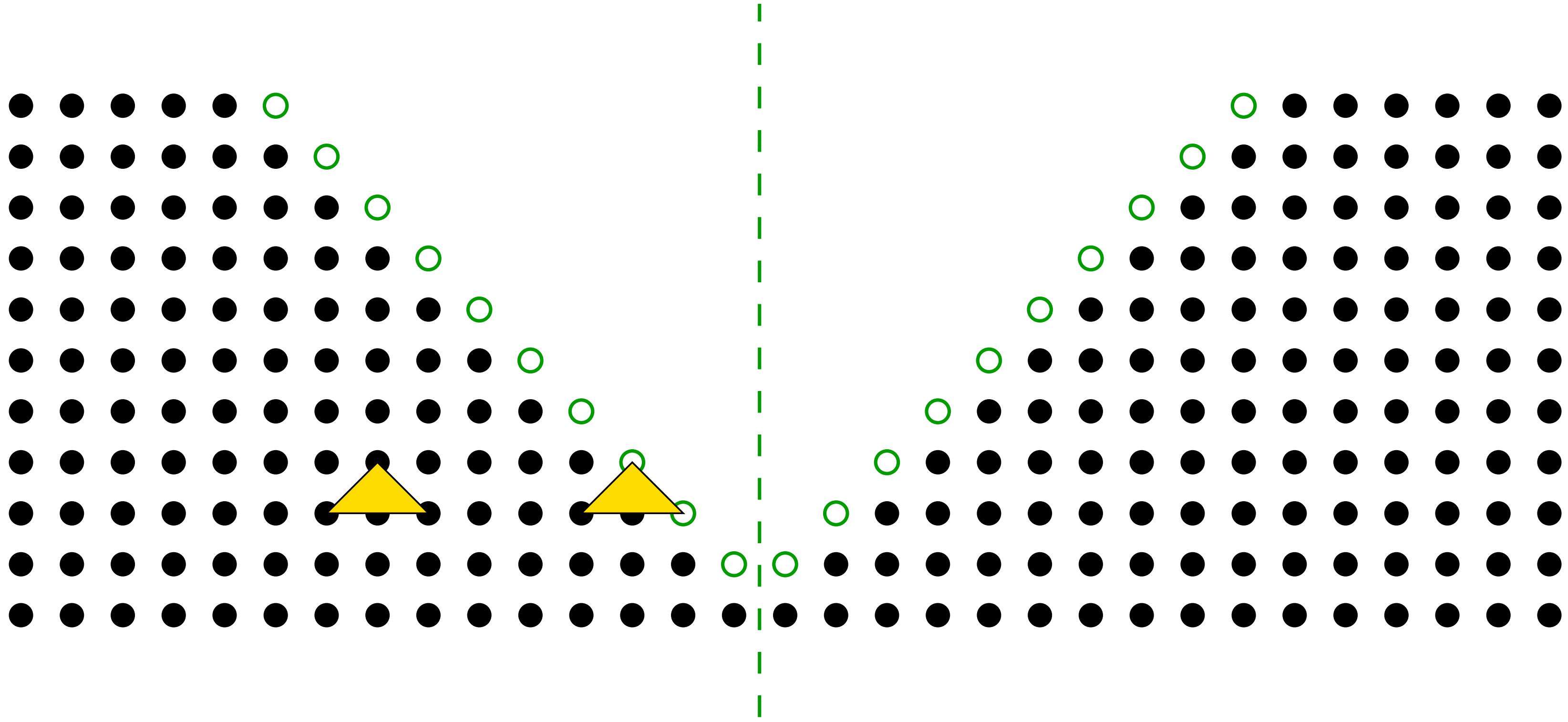
# Using Extra Computation in Time Dependent Problems

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

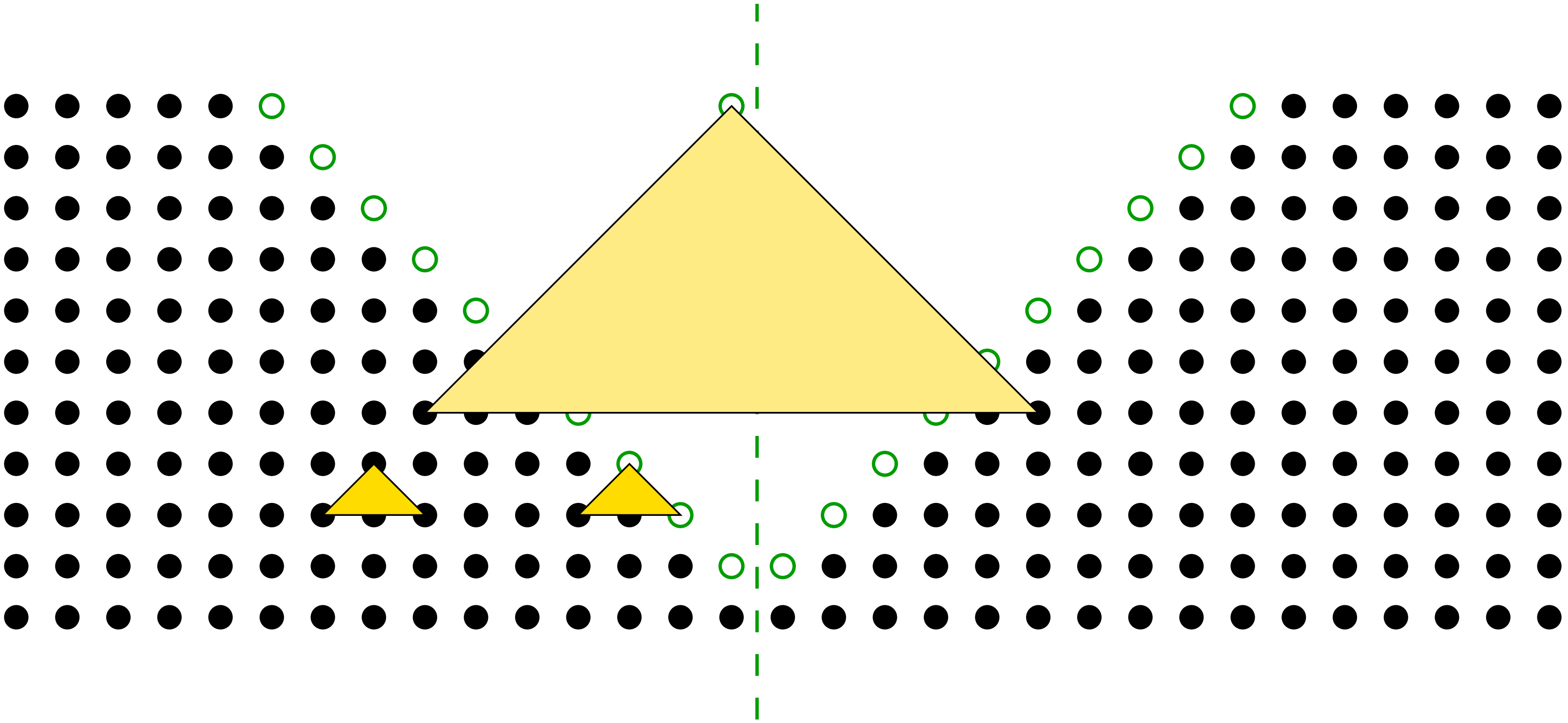
# Trading Computation for Communication

- 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

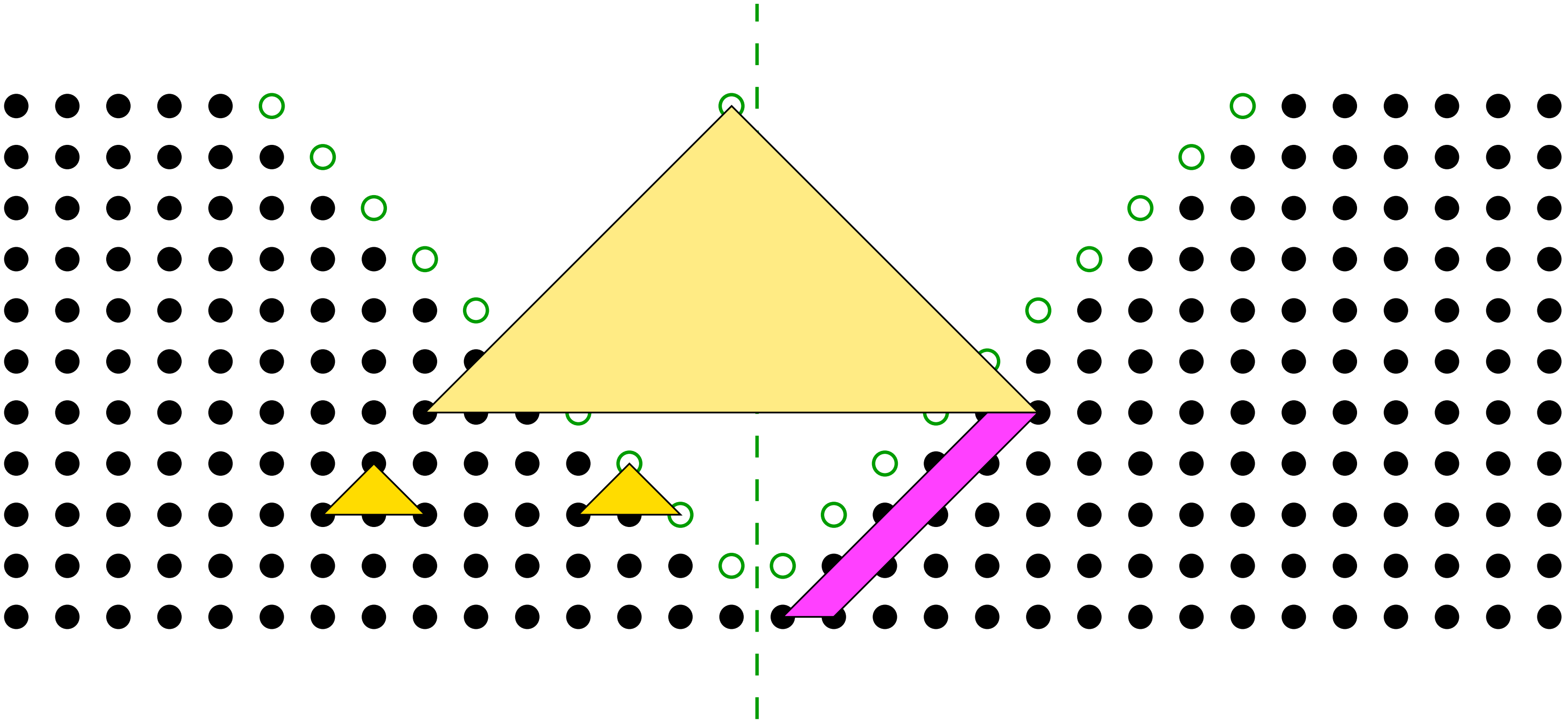


# 1-D Time Stepping

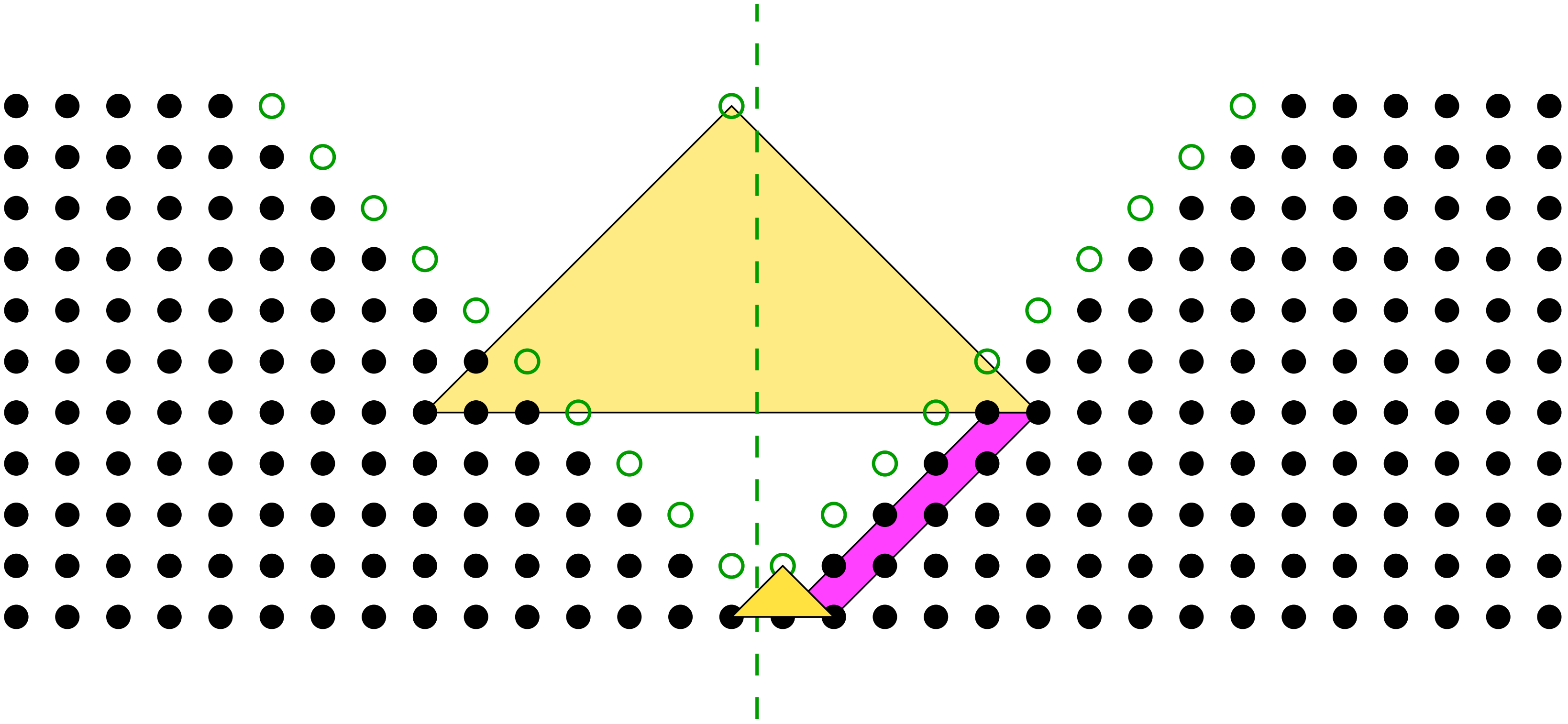


PARALLEL@ILLINOIS

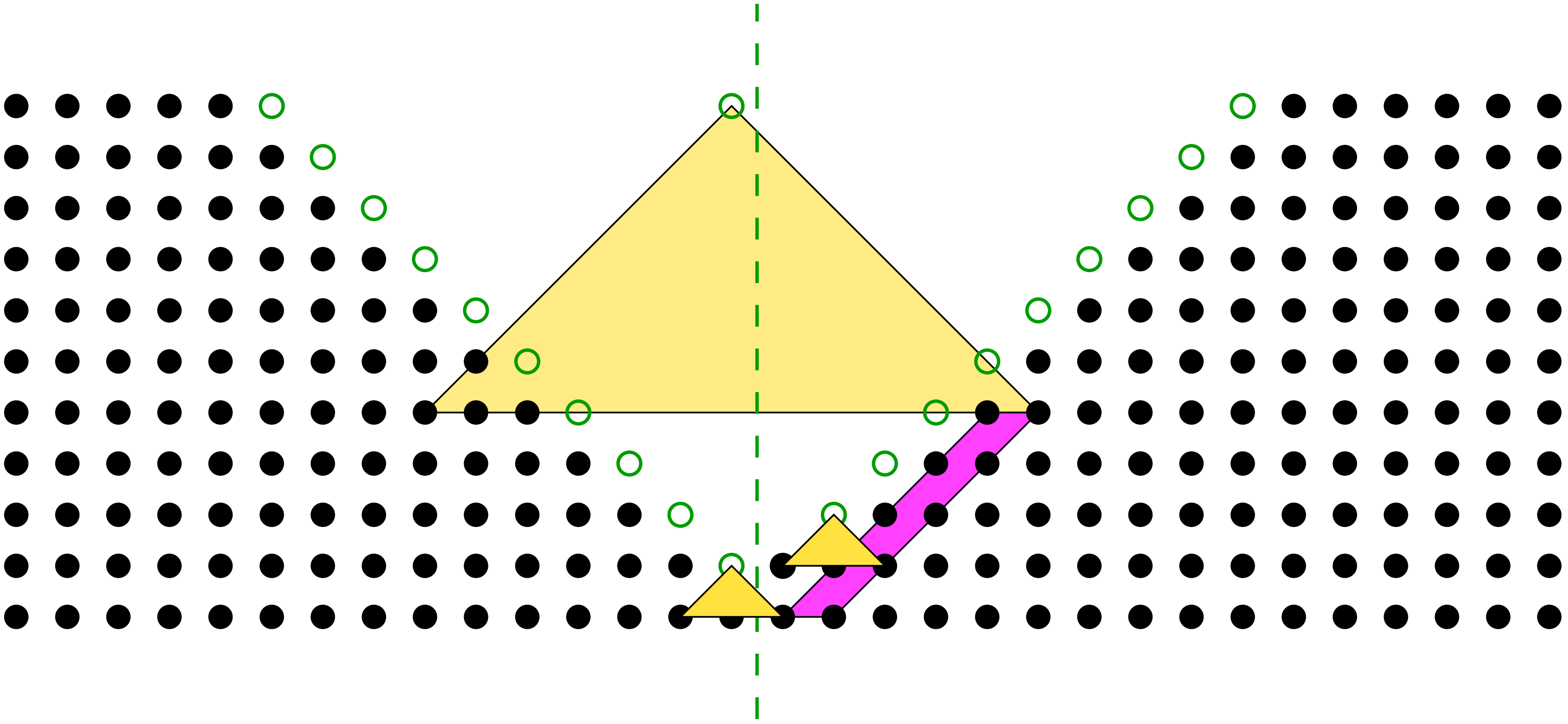
# 1-D Time Stepping



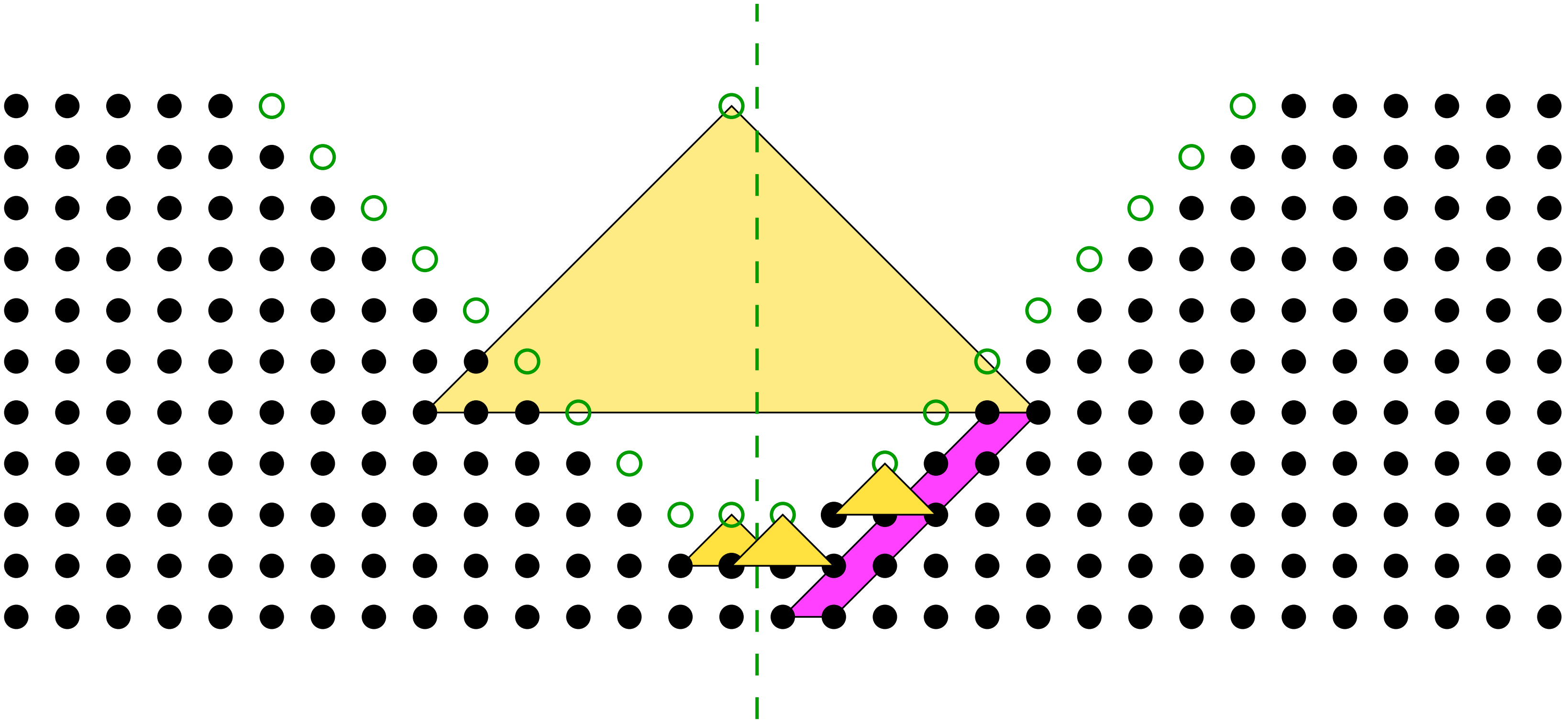
# 1-D Time Stepping



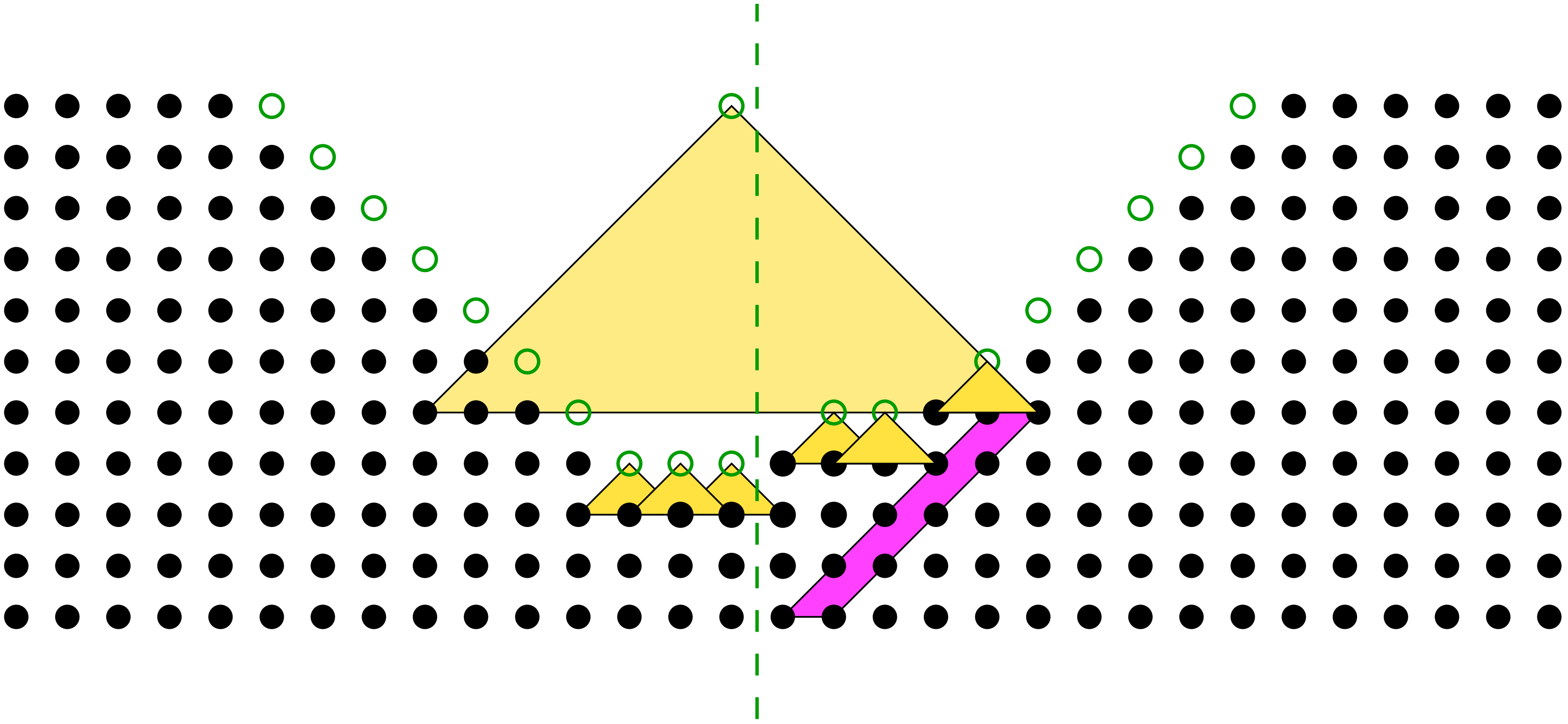
# 1-D Time Stepping



# 1-D Time Stepping



# 1-D Time Stepping



# Analyzing the Cost of Redundant Computation

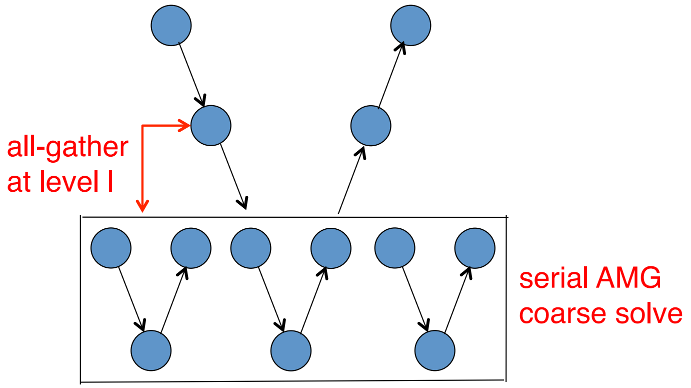
- Advantage of redundant computation:
  - ♦ Communication costs:
    - K steps, 1 step at a time: 2k(s+w)
    - K steps at once: 2(s+kw)
  - ♦ Redundant computation is roughly
    - $Ak^2c$, for A operations for each eval and time c for each operation
- Thus, redundant computation better when
  - ♦ $Ak^2c < 2(k-1)s$
- 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,
  - ♦ $u^{K+2} = Au^{k+1} = Aau^k = A^2u^k$
  - ♦ And so on
- Thus, this approach can be used to efficiently compute
  - ♦ $Ax$, $A^2x$, $A^3x$, …
- 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 (more later)?

# Using Redundant Solvers

- AMG requires a solve on the coarse grid



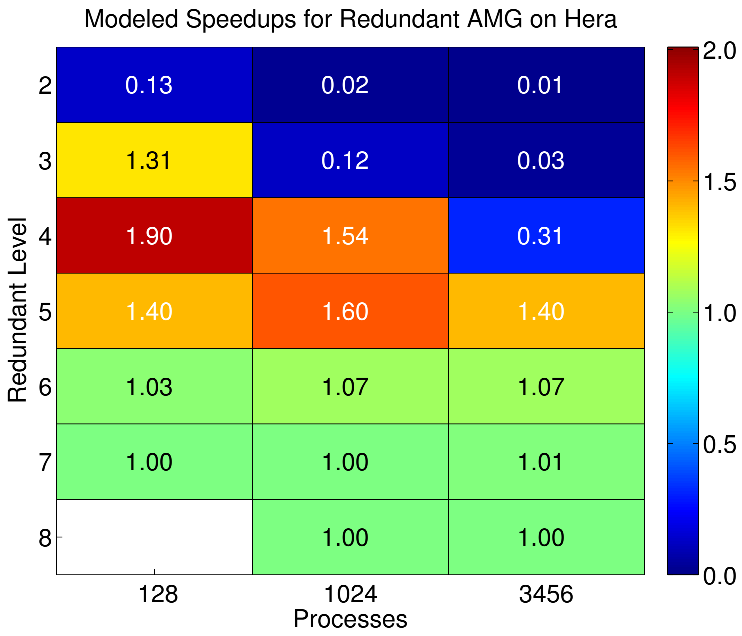
- Rather than either solve in parallel (too little work for the communication) or solve in serial and distribute solution, solve redundantly (either in smaller parallel groups or serial, as in this illustration)

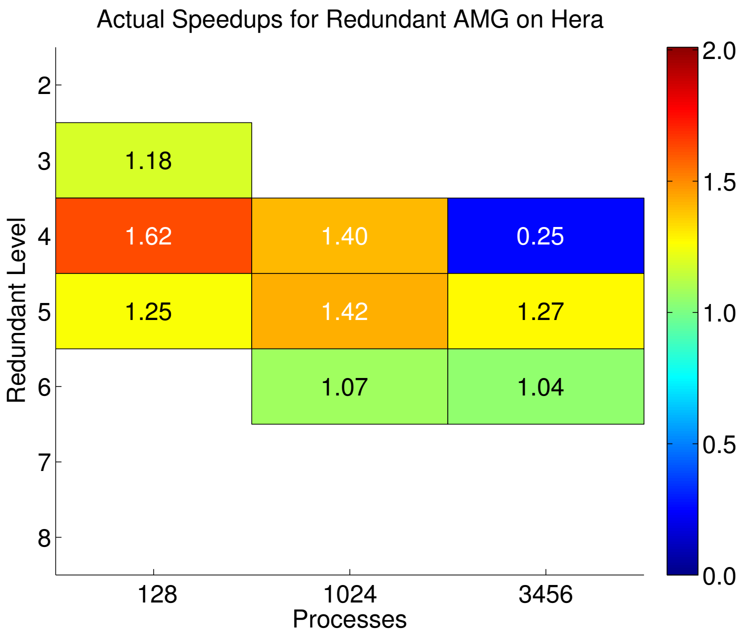
# Redundant Solution

- Replace communication at levels ≥lred with Allgather
- Every process now has complete information; no further communication needed
- Performance analysis (based on Gropp & Keyes 1989) can guide selection of lred

# Redundant Solves

- Applied to Hera at LLNL, provides significant speedup


Modeled Speedups for Redundant AMG on Hera


Actual Speedups for Redundant AMG on Hera

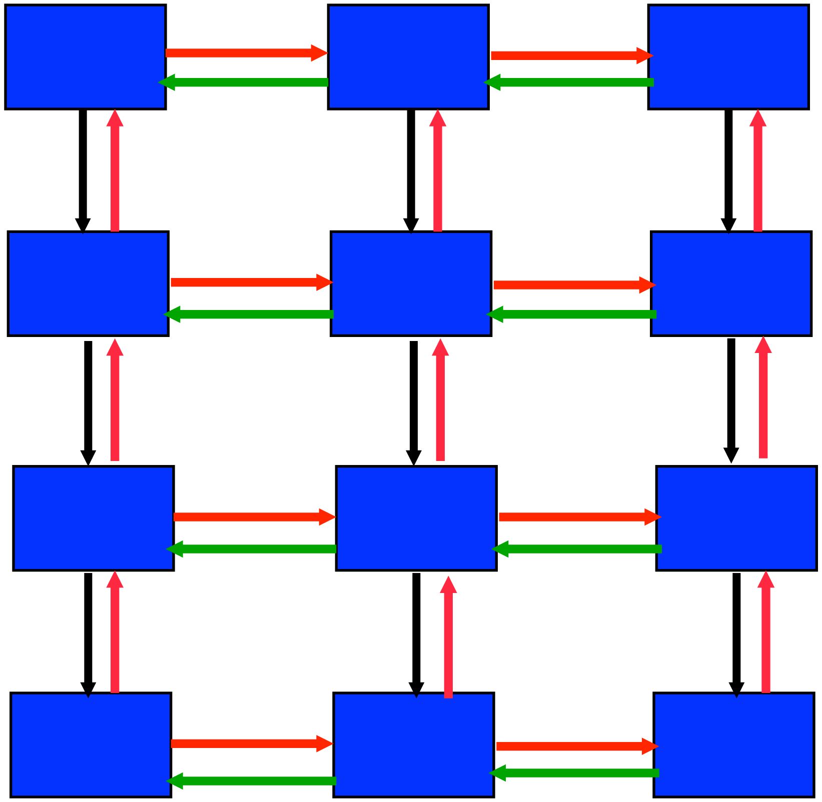
- Thanks to Hormozd Gahvari

# Contention and Communication

- The simple model $T = s + rn$ (and slightly more detailed logp models) can give good guidance but ignores some important effects
- One example is regular mesh halo exchange

# Mesh Exchange

- Exchange data on a mesh



# Nonblocking Receive and Blocking Sends

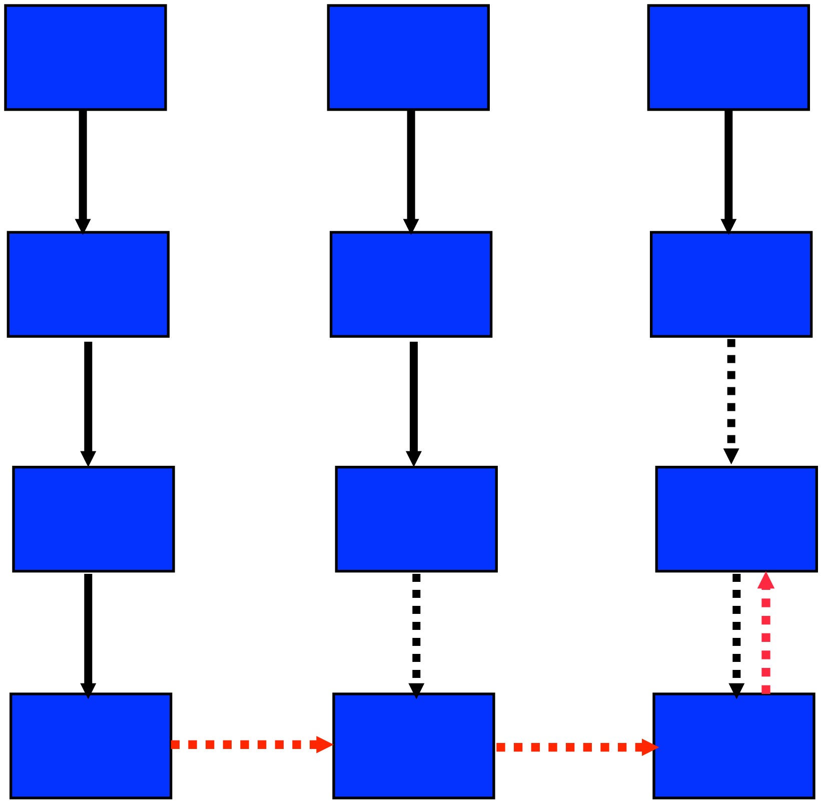
-
  ```
  Do i=1,n_neighbors
     Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag,&
                             comm, requests(i), ierr)
  Enddo
  Do i=1,n_neighbors
     Call MPI_Send(edge(1,i), len, MPI_REAL, nbr(i), tag,&
                             comm, ierr)
  Enddo
  Call MPI_Waitall(n_neighbors, requests, statuses, ierr)
  ```
- Does not perform well in practice (at least on BG, SP). Why?

# Understanding the Behavior: Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

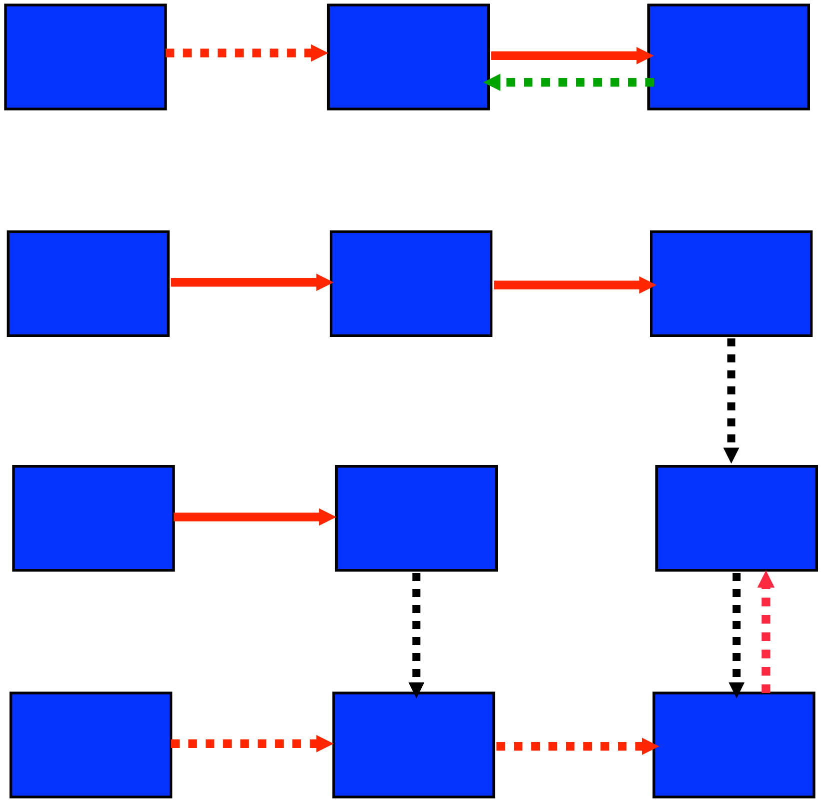
# Mesh Exchange - Step 1

- Exchange data on a mesh



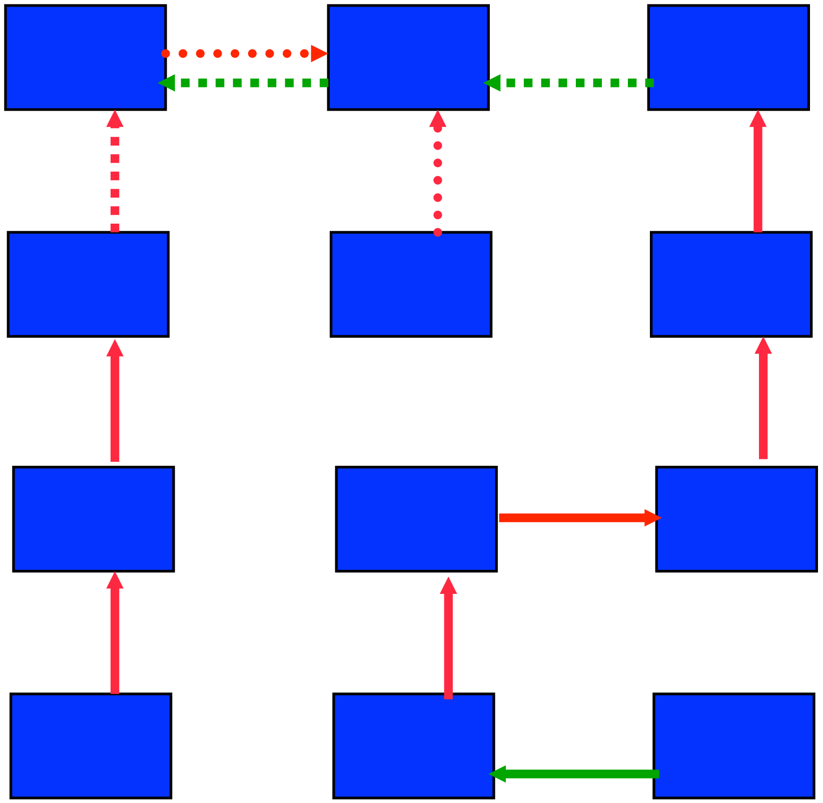
# Mesh Exchange - Step 2

- Exchange data on a mesh



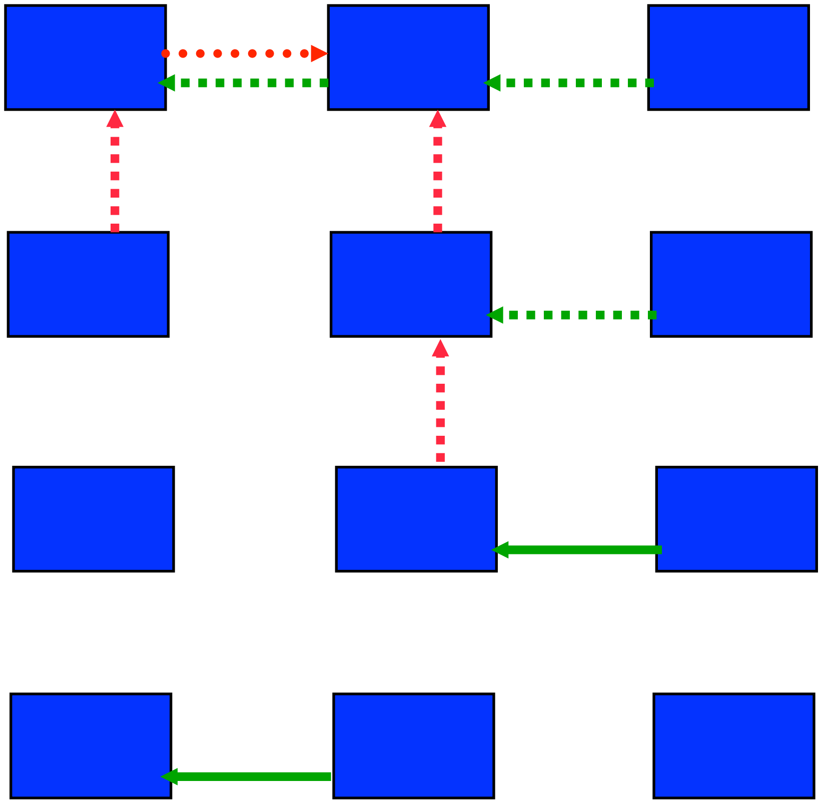
# Mesh Exchange - Step 3

- Exchange data on a mesh



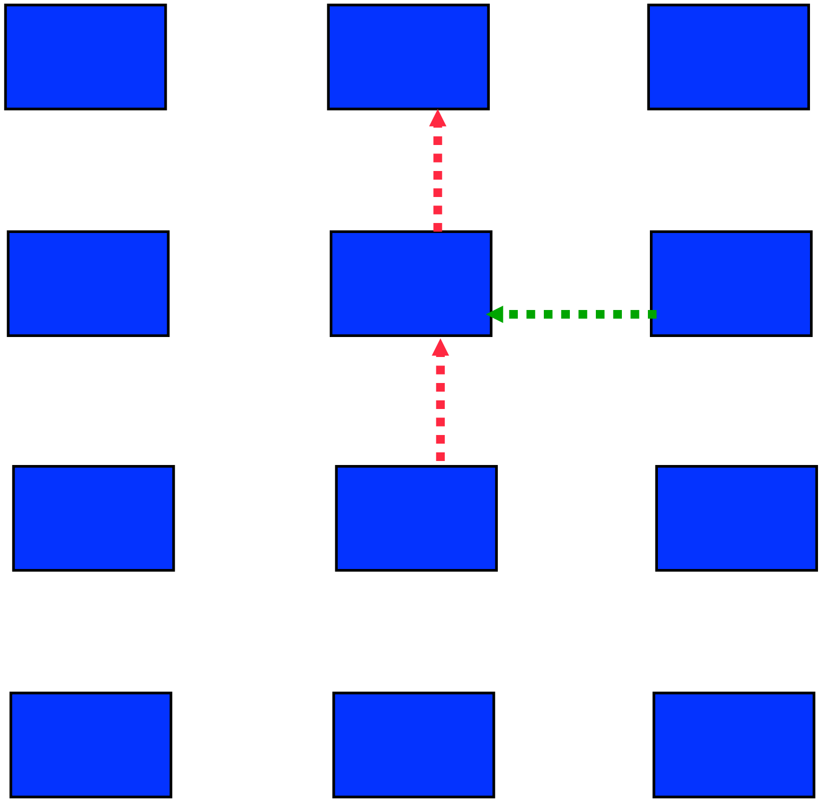
# Mesh Exchange - Step 4

- Exchange data on a mesh



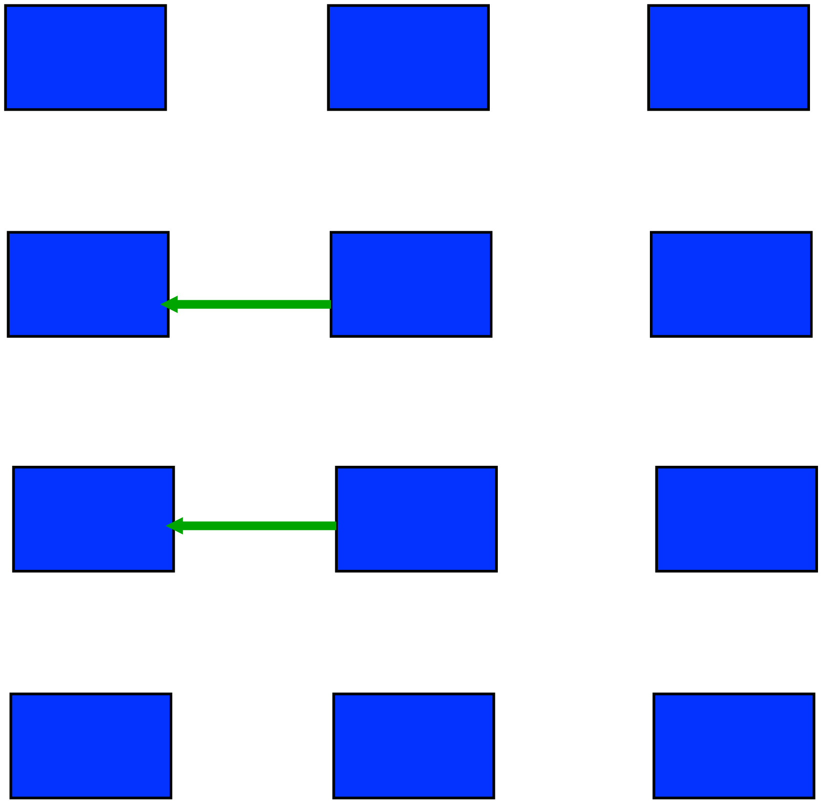
# Mesh Exchange - Step 5

- Exchange data on a mesh

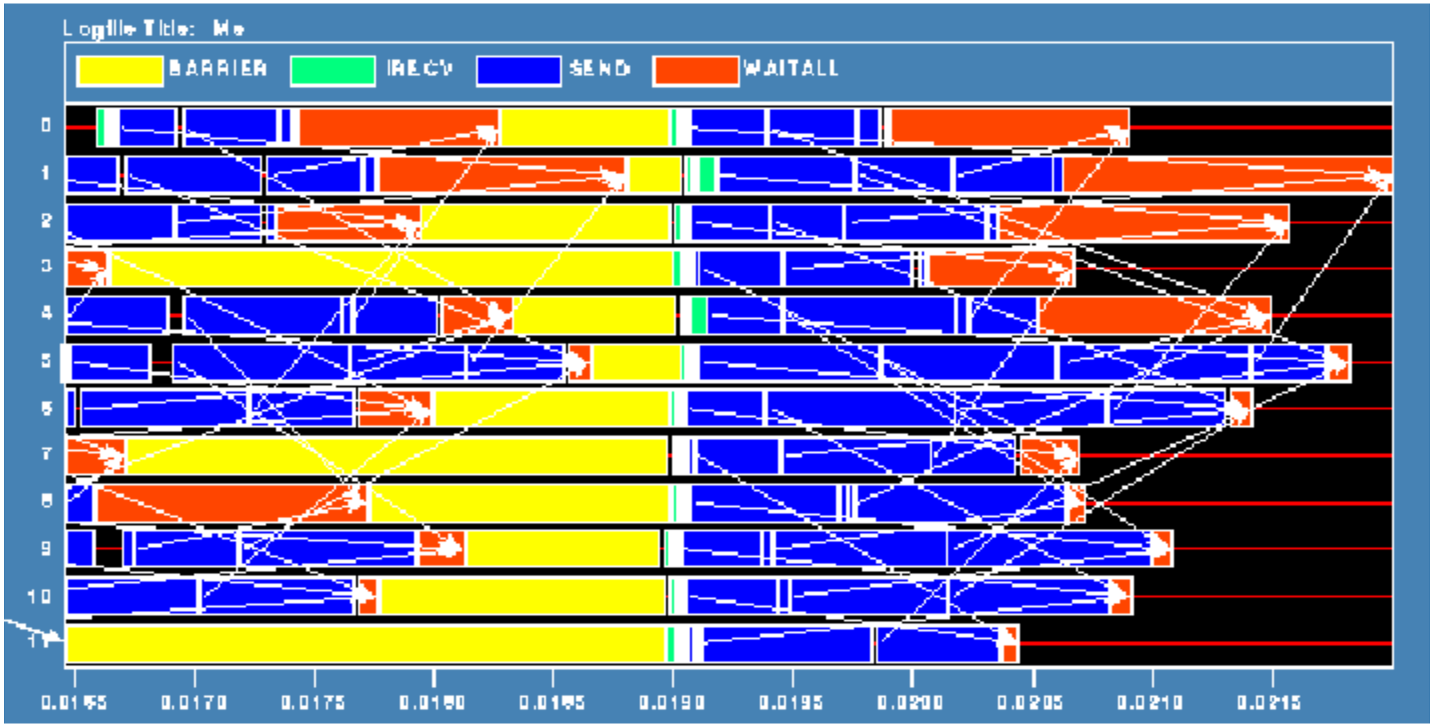


# Mesh Exchange - Step 6

- Exchange data on a mesh

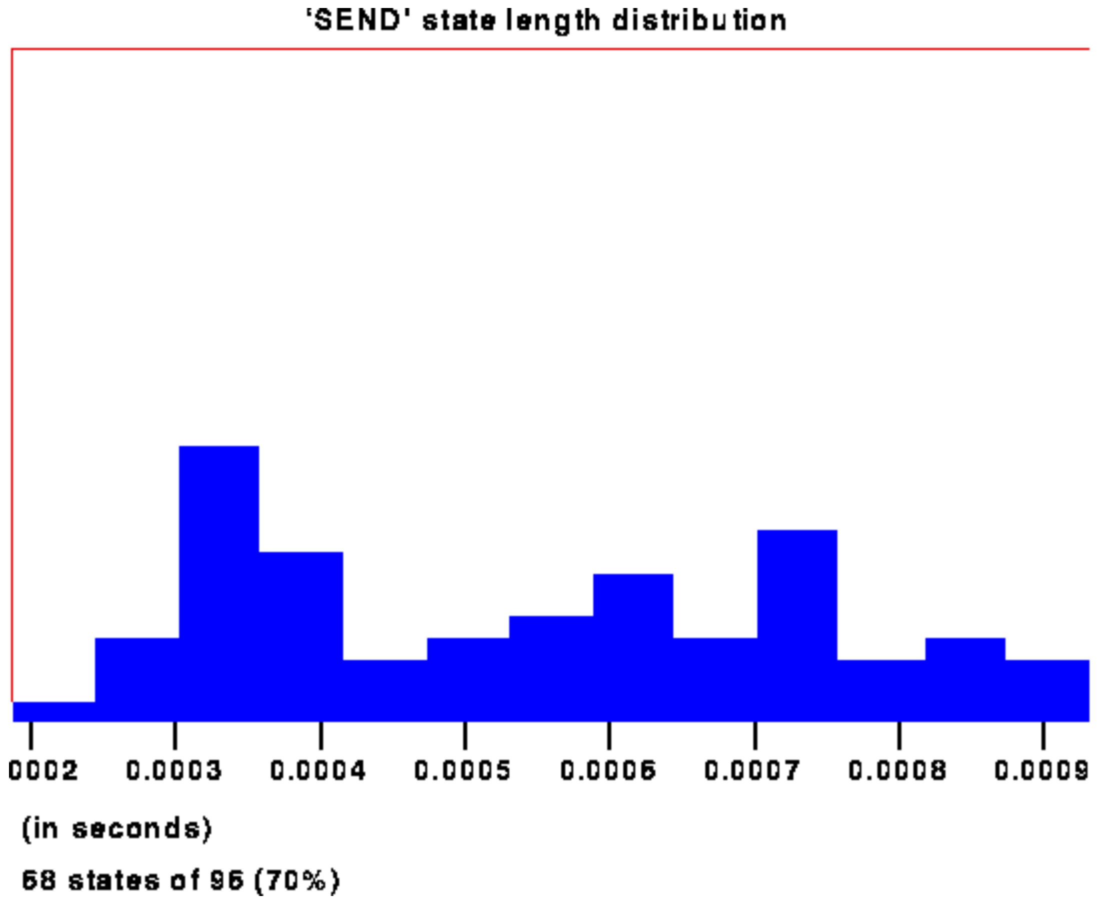


# Timeline from IBM SP



- Note that process 1 finishes last, as predicted

# Distribution of Sends

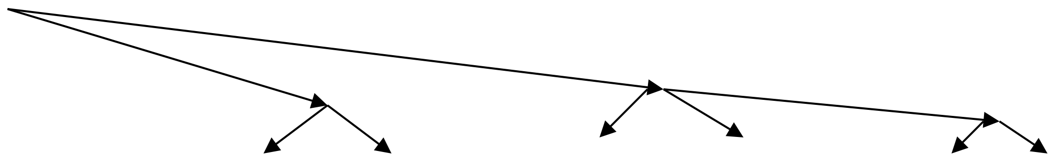


# Why Six Steps?

- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory
- The interference of communication is why adding an MPI_Barrier (normally an unnecessary operation that reduces performance) can occasionally increase performance.  But don’t add MPI_Barrier to your code, please :)

# Thinking about Broadcasts

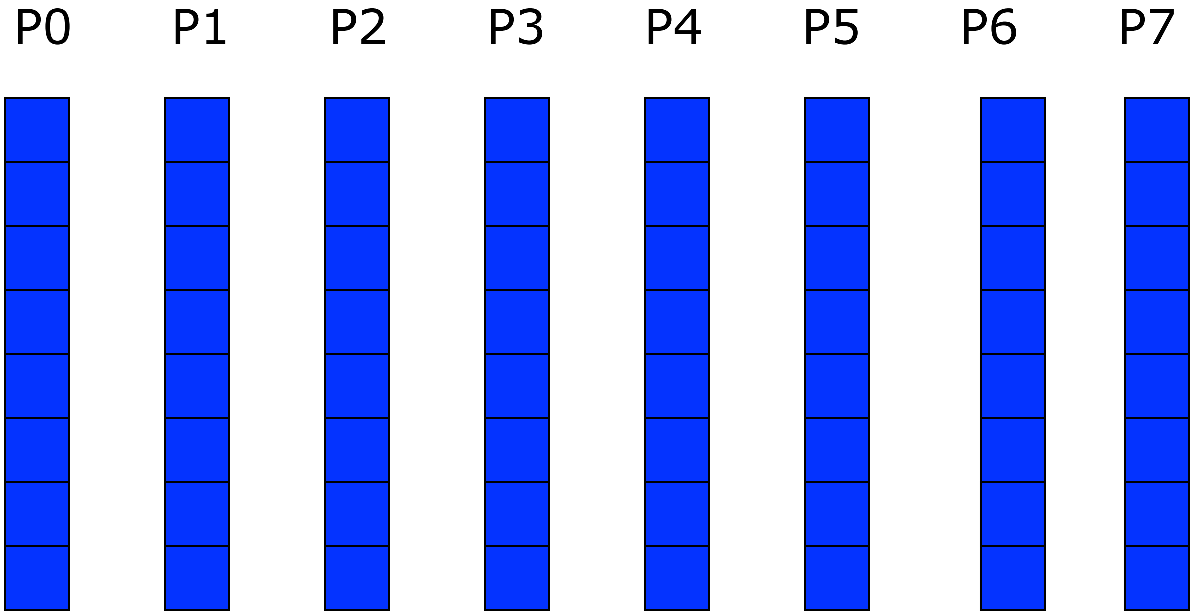
- MPI_Bcast( buf, 100000, MPI_DOUBLE, … );
- Use a tree-based distribution:



- Use a *pipeline*: send the message in b byte pieces. This allows each subtree to begin communication after b bytes sent
- Improves total performance:
  - Root process takes same time (asymptotically)
  - Other processes wait less
    - Time to reach leaf is *b log p + (n-b)*, rather than *n log p*
- Special hardware and other algorithms can be used …

# Make Full Use of the Network

- Implement MPI_Bcast(buf,n,…) as
  MPI_Scatter(buf, n/p,…, buf+rank*n/p,…)
  MPI_Allgather(buf+rank*n/p, n/p,…,buf,…)



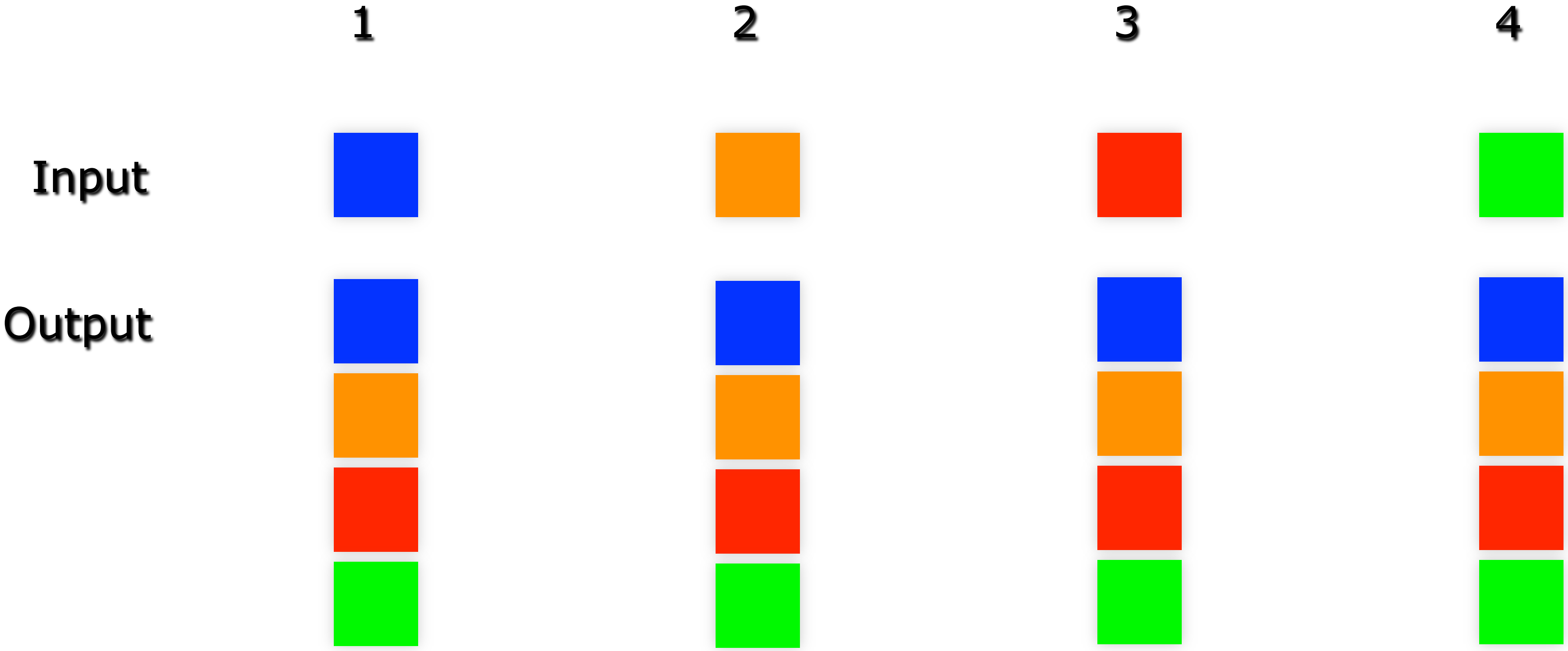
# Optimal Algorithm Costs

- Optimal cost is O(n) (O(p) terms don't involve n) since scatter moves n data, and allgather also moves only n per process; these can use pipelining to move data as well
  - ♦ Scatter by recursive bisection uses log p steps to move n(p-1)/p data
  - ♦ Scatter by direct send uses p-1 steps to move n(p-1)/p data
  - ♦ Recursive doubling allgather uses log p steps to move
    - N/p + 2n/p + 4n/p + … (p/2)/p = n(p-1)/p
  - ♦ Bucket brigade allgather moves
    - N/p (p-1) times or (p-1)n/p
- See, e.g., van de Geijn for more details

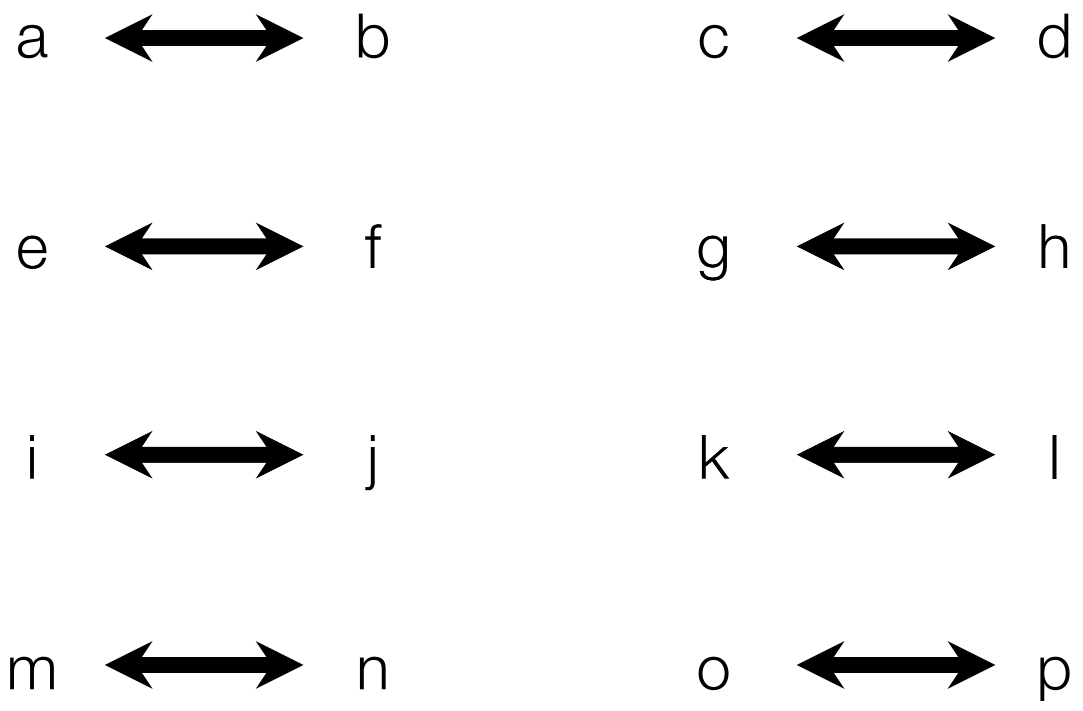
# Is it communication avoiding or minimum solution time?

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

# Allgather



# Allgather: recursive doubling



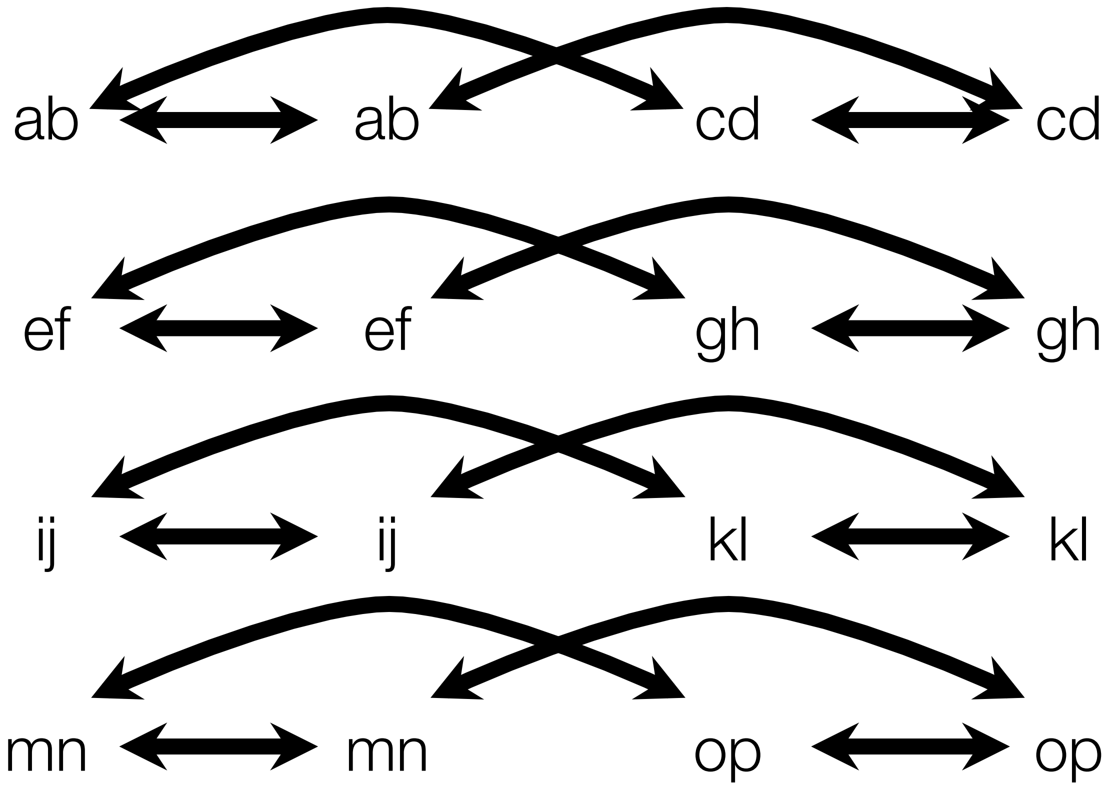
a ↔ b c ↔ d

e ↔ f g ↔ h

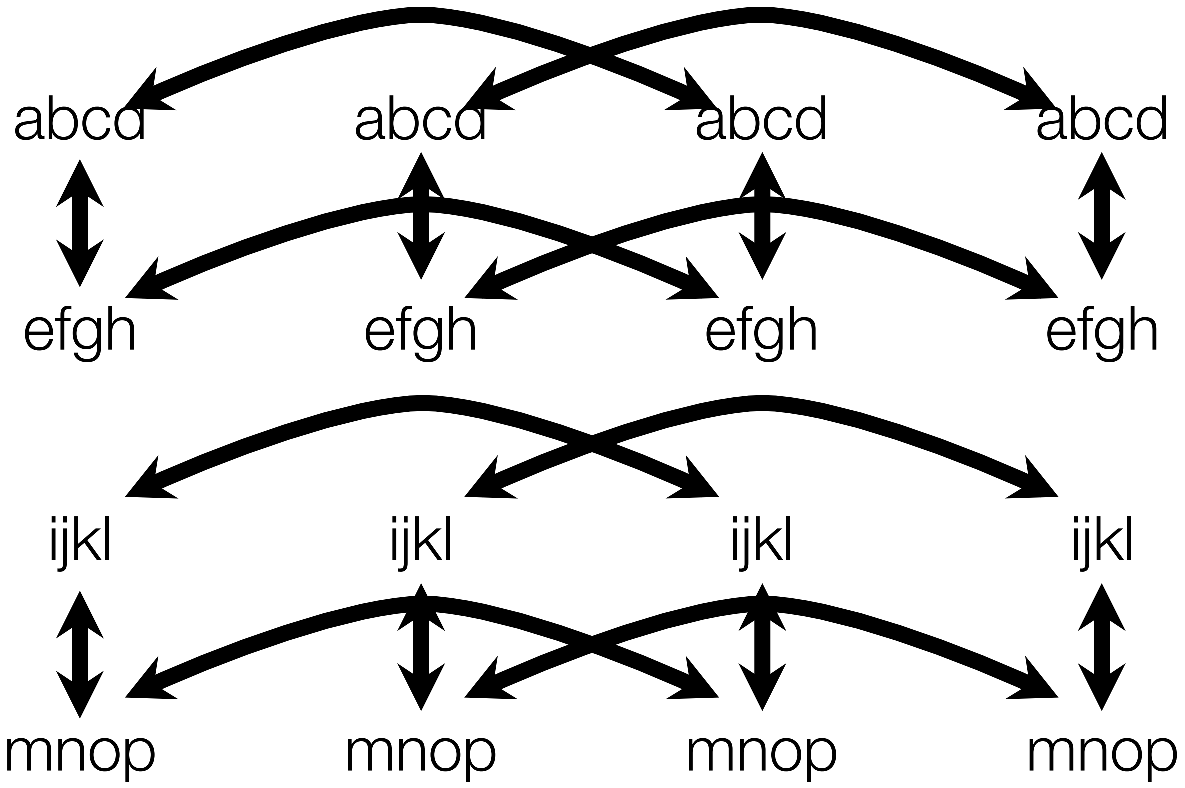
i ↔ j k ↔ l

m ↔ n o ↔ p

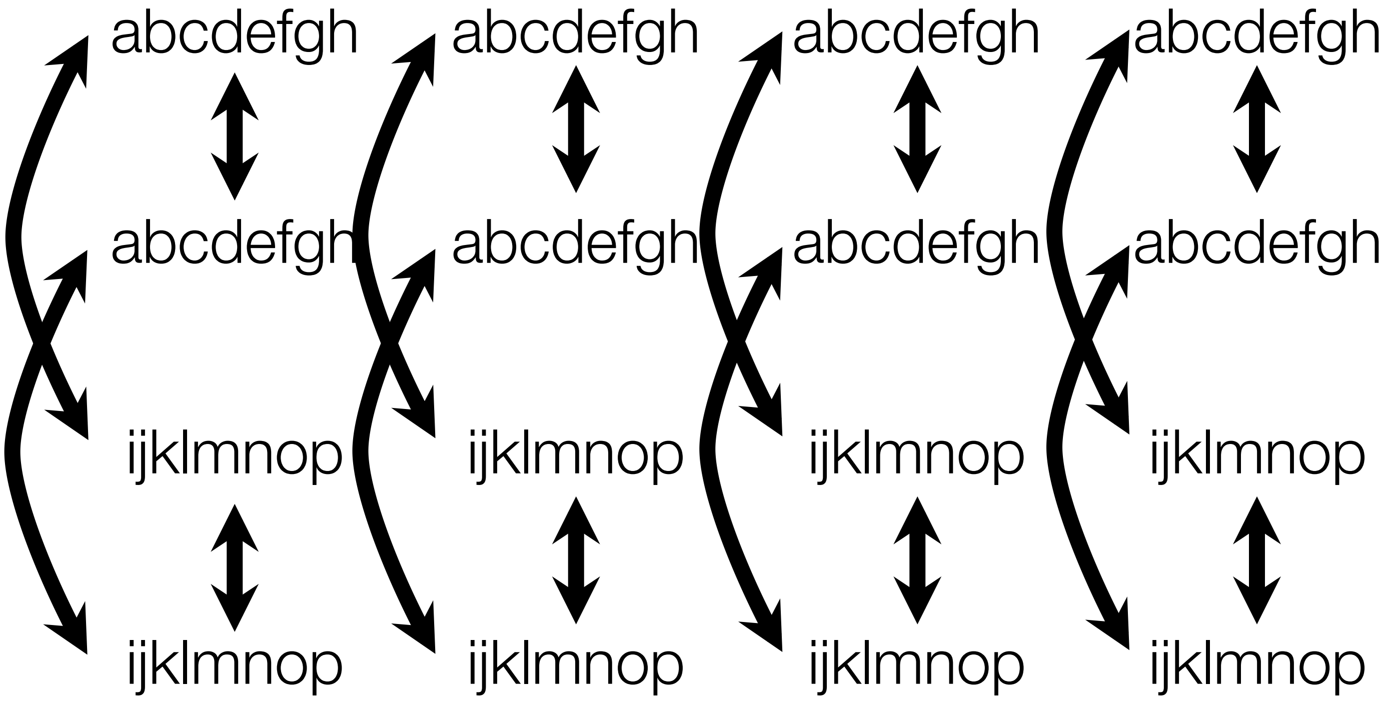
# Allgather: recursive doubling



# Allgather: recursive doubling



# Allgather: recursive doubling



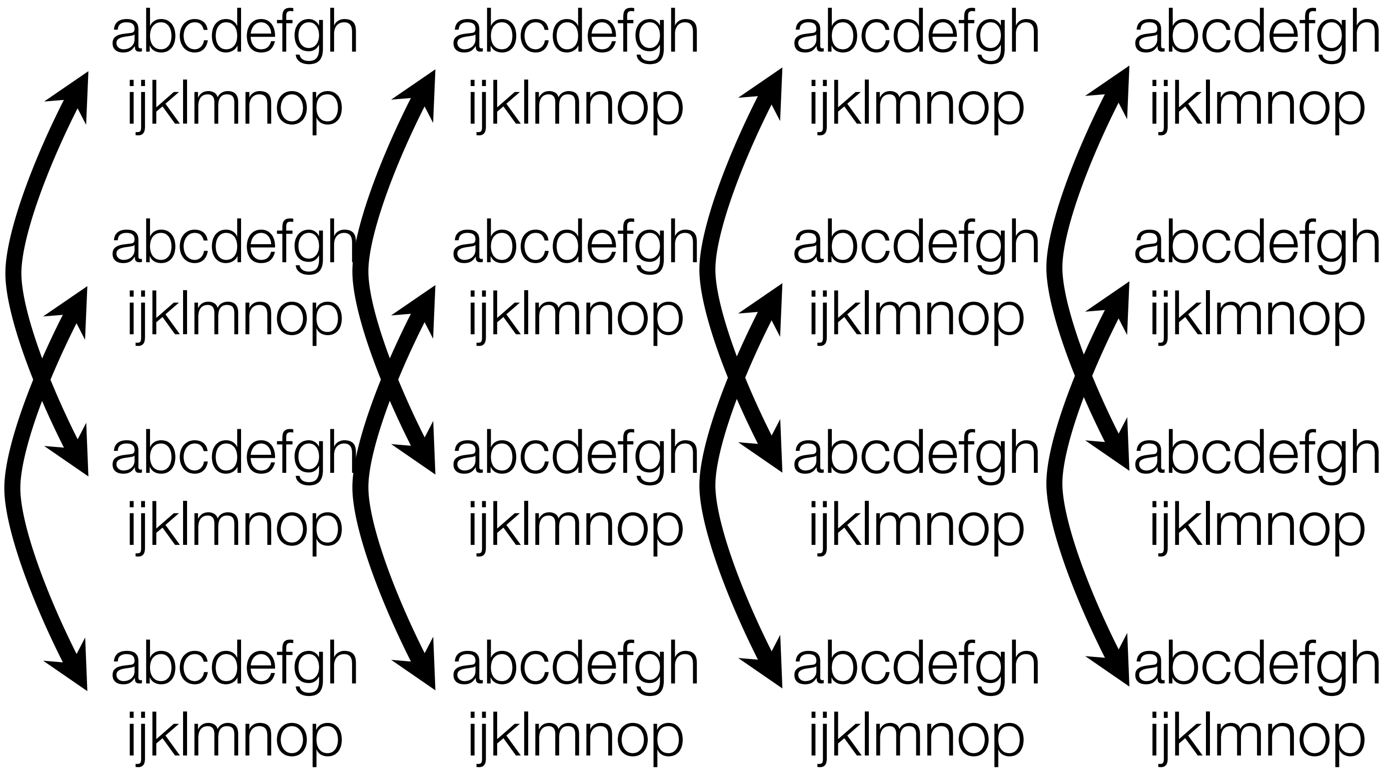
abcdefgh abcdefgh abcdefgh abcdefgh

abcdefgh abcdefgh abcdefgh abcdefgh

ijklmnop ijklmnop ijklmnop ijklmnop

ijklmnop ijklmnop ijklmnop ijklmnop

# Allgather: recursive doubling



abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop

abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop

abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop

abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop abcdefgh ijklmnop

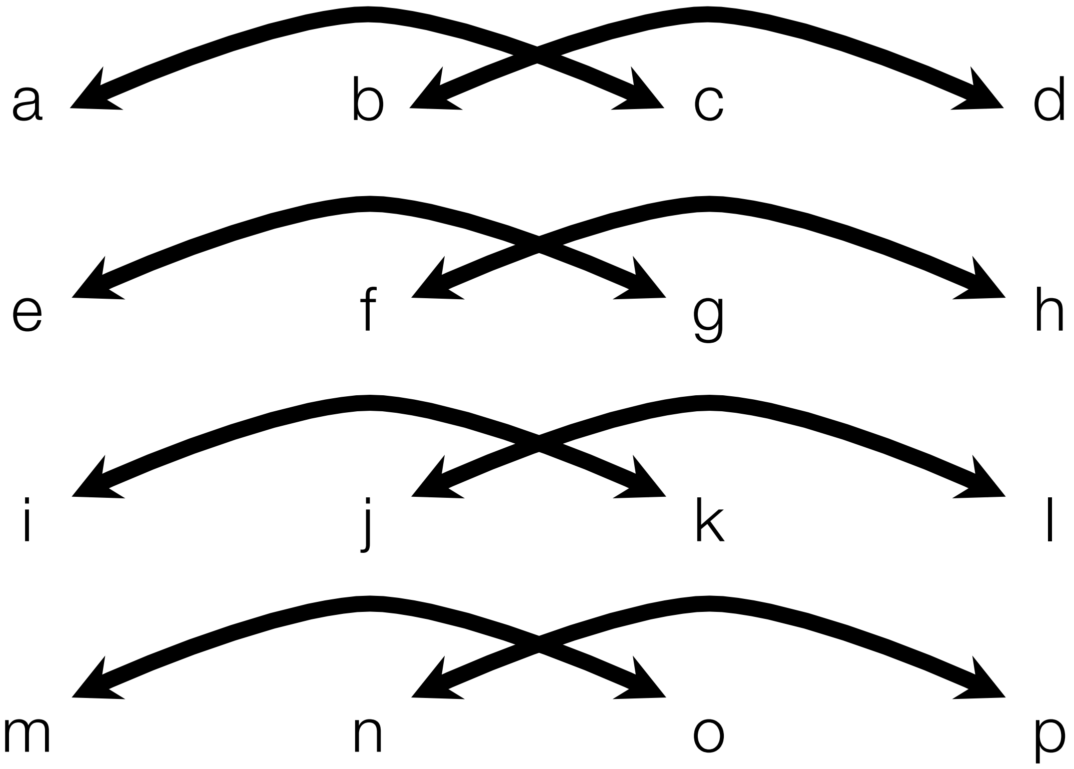
$$T=(\mathit{lg}\ P)\ \alpha + n(P-1)\beta$$

# Problem: Recursive-doubling

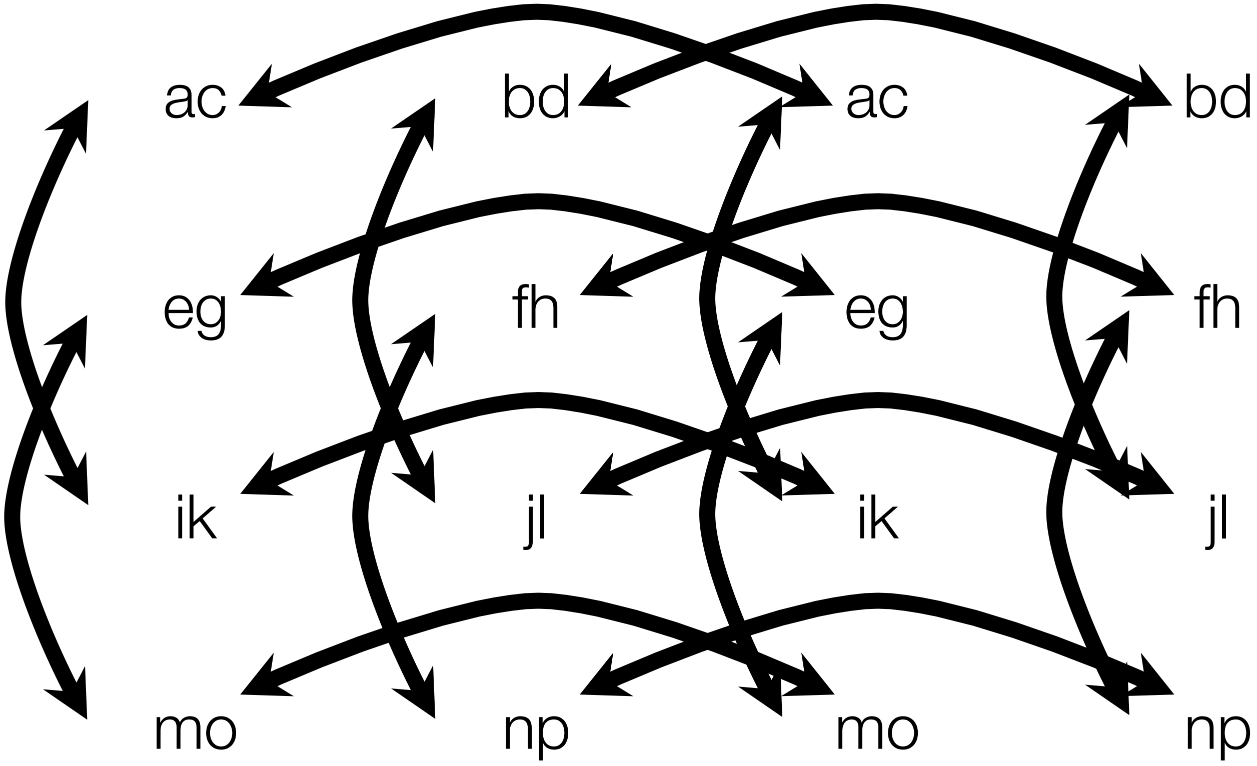
- No congestion model:
  - ♦ $T=(\lg P)\alpha + n(P-1)\beta$
- Congestion on torus:
  - ♦ $T\approx(\lg P)\alpha + (5/24)nP^{4/3}\beta$
- Congestion on Clos network:
  - ♦ $T\approx(\lg P)\alpha + (nP/\mu)\beta$

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

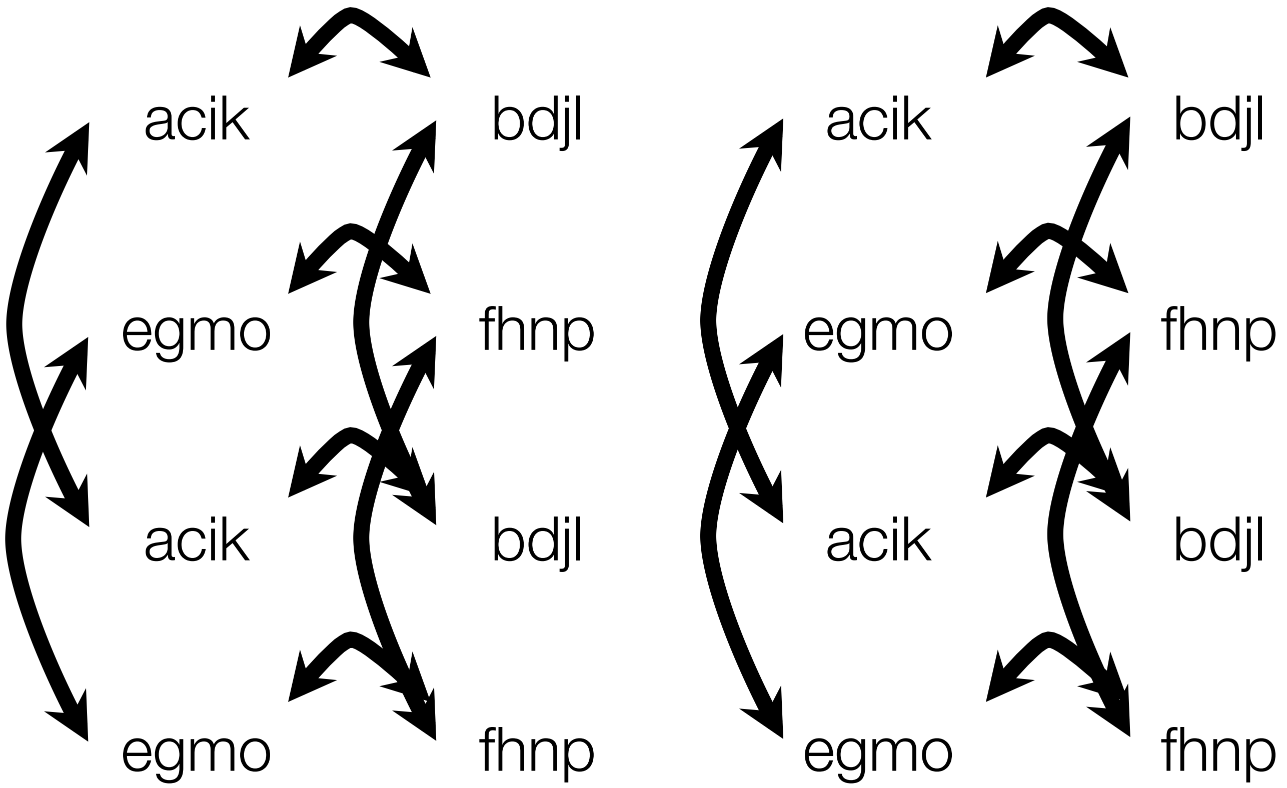
# Allgather: recursive halving



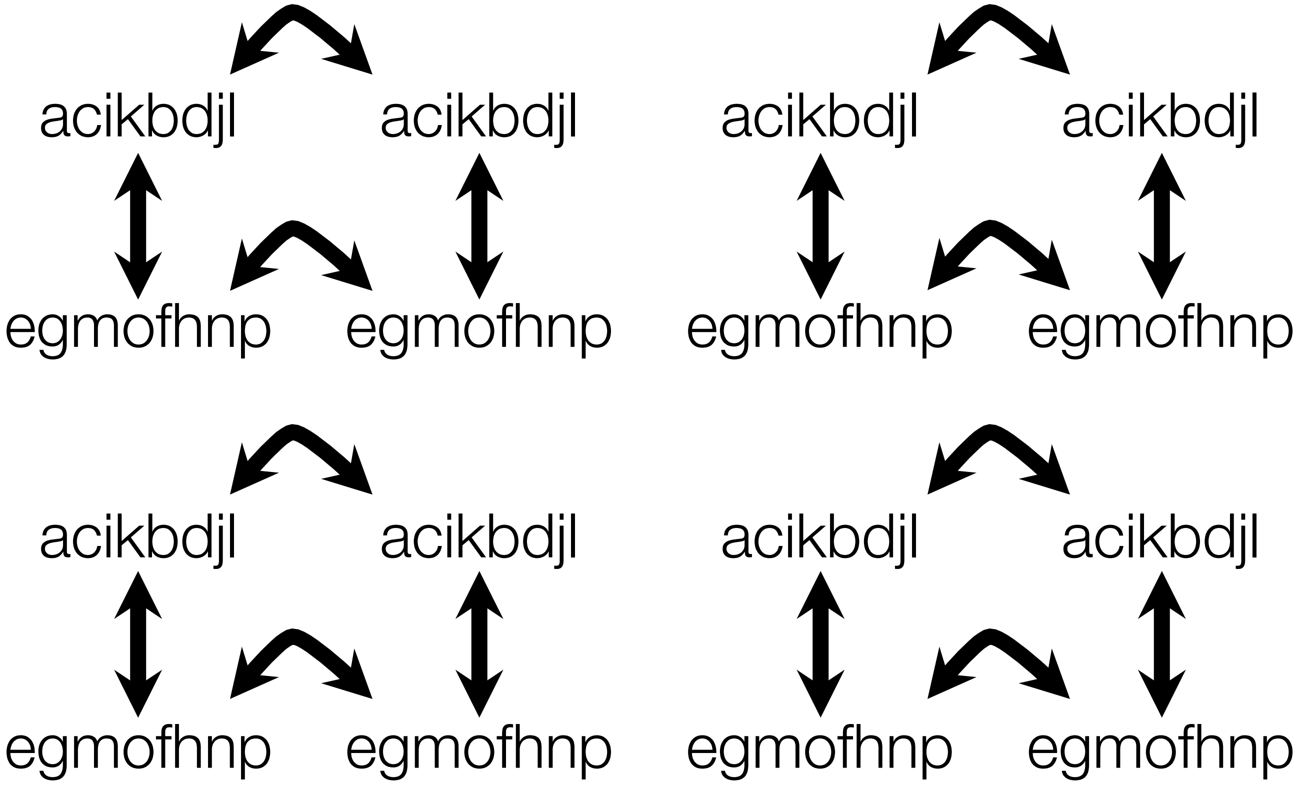
# Allgather: recursive halving



# Allgather: recursive halving



# Allgather: recursive halving



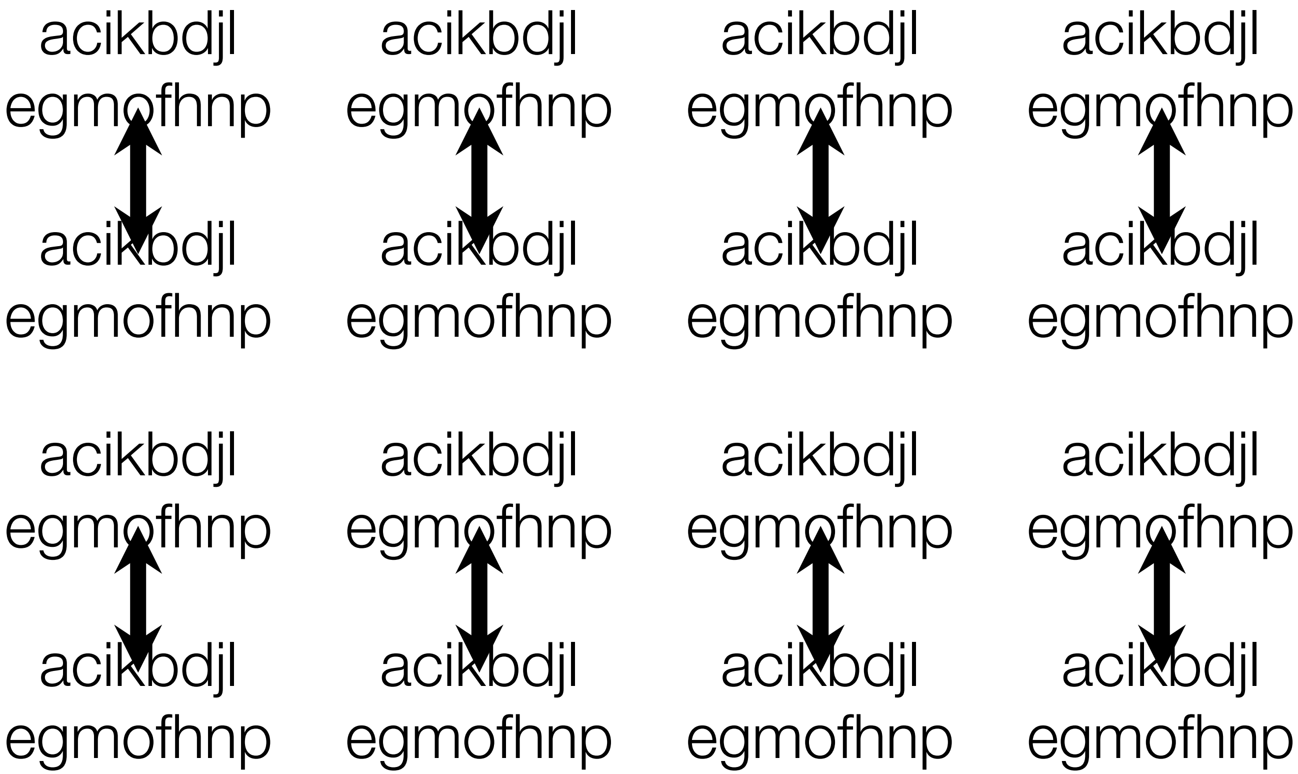
acikbdjl acikbdjl acikbdjl acikbdjl

egmofhnp egmofhnp egmofhnp egmofhnp

acikbdjl acikbdjl acikbdjl acikbdjl

egmofhnp egmofhnp egmofhnp egmofhnp

# Allgather: recursive halving



acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp

acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp

acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp

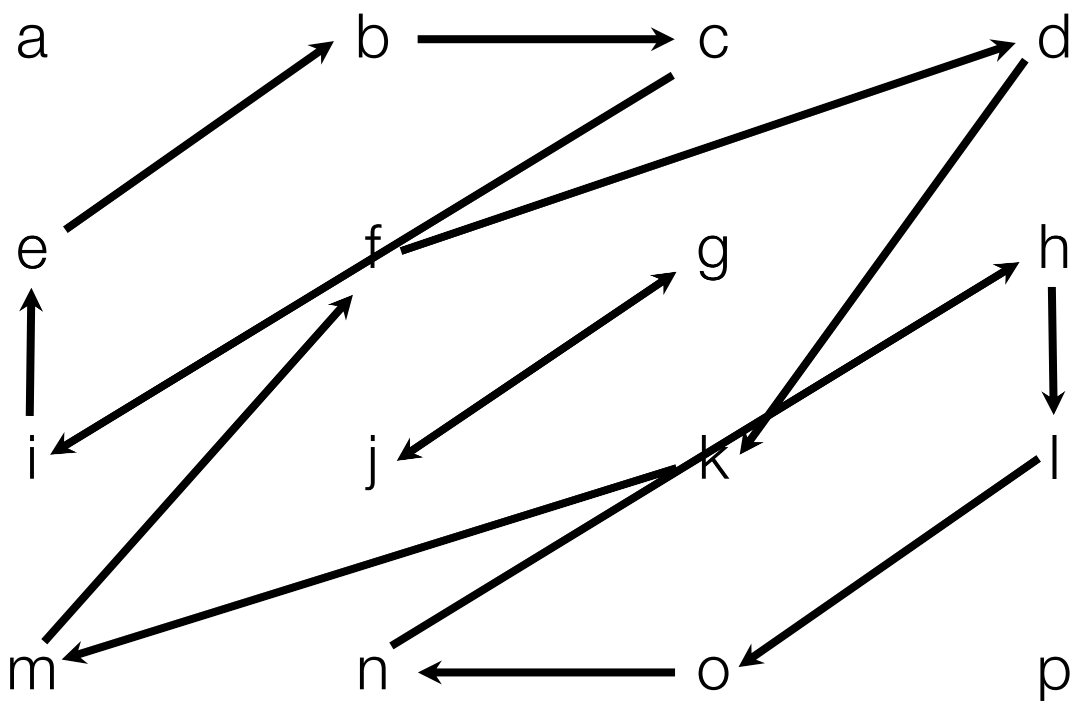
acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp | acikbdjl egmofhnp

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

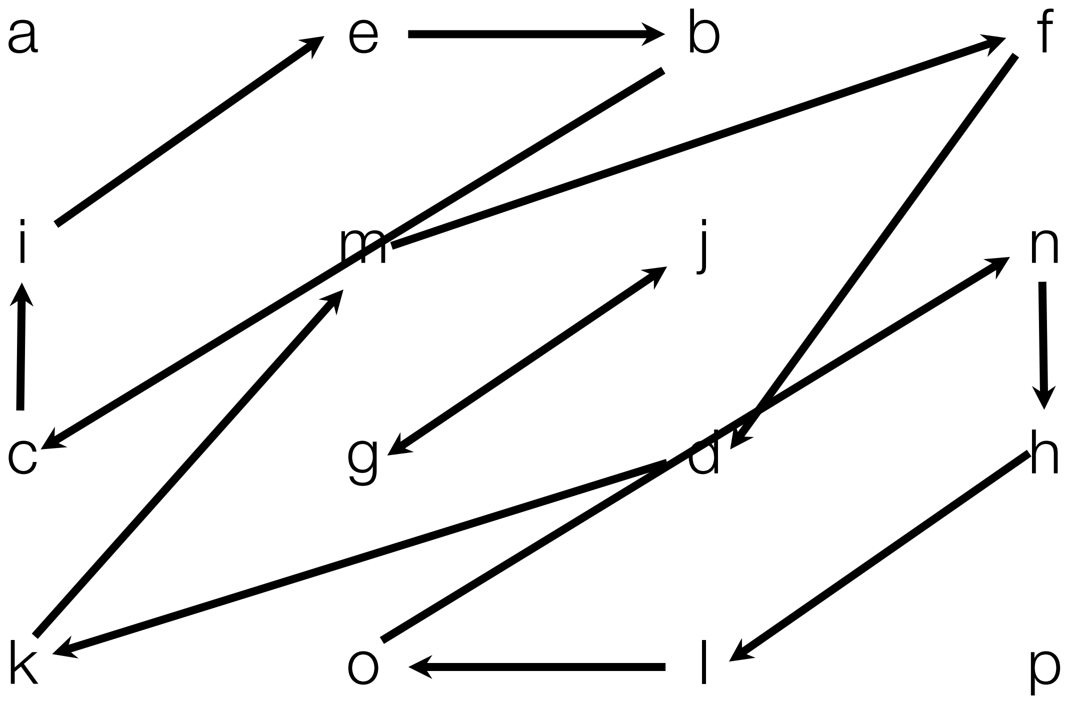
# New problem: data misordered

- Solution: shuffle input data
    - ♦ Could shuffle at end (redundant work; all processes shuffle)
    - ♦ Could use non-contiguous data moves
    - ♦ Shuffle data on network…

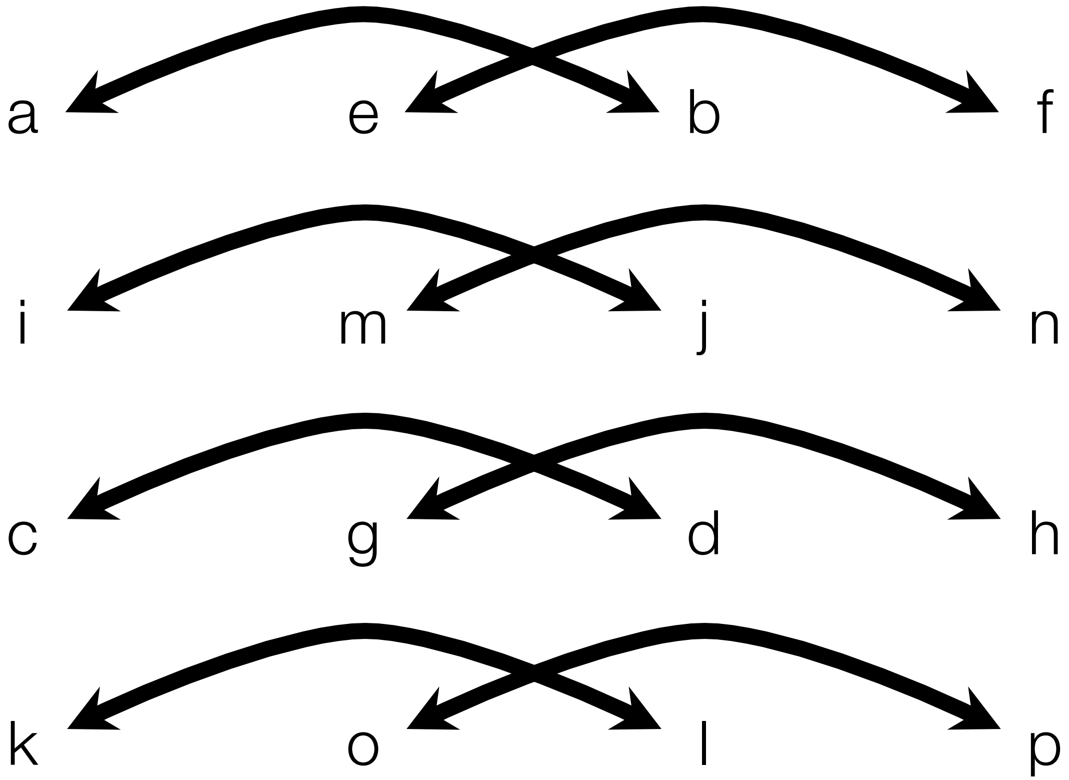
# Solution: Input shuffle



# Solution: Input shuffle



# Solution: Input shuffle



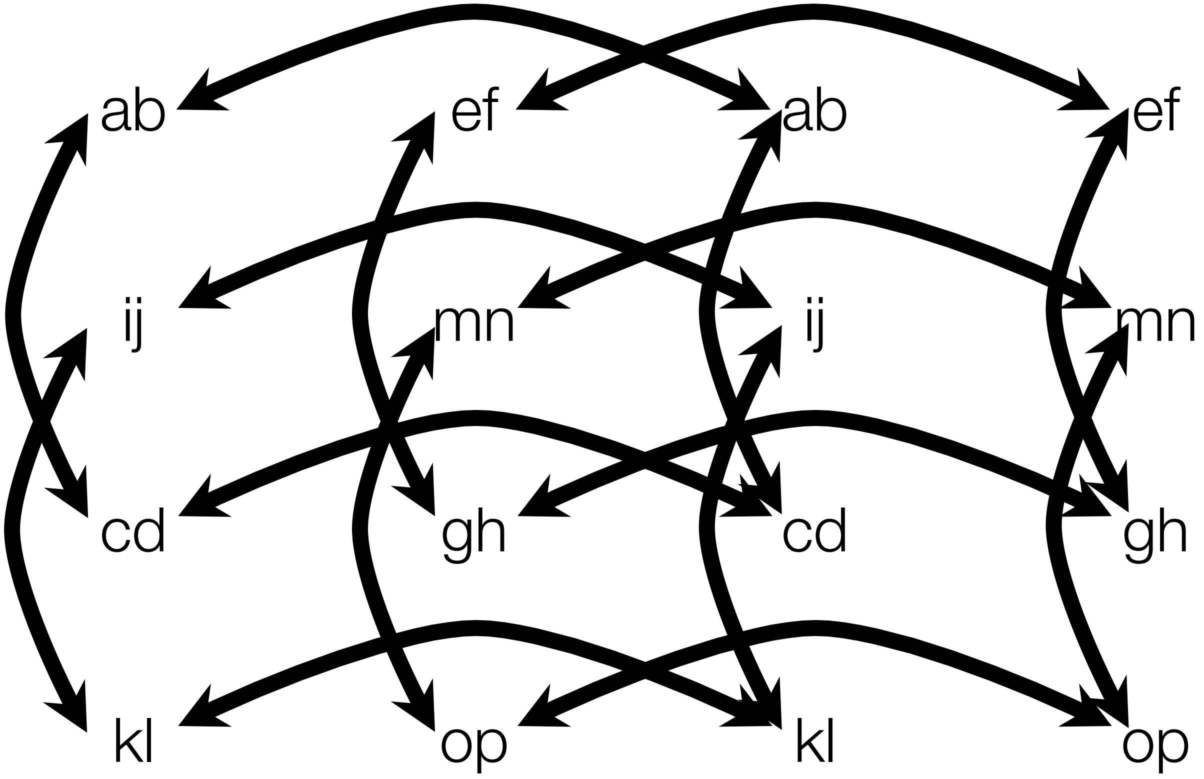
a e b f

i m j n

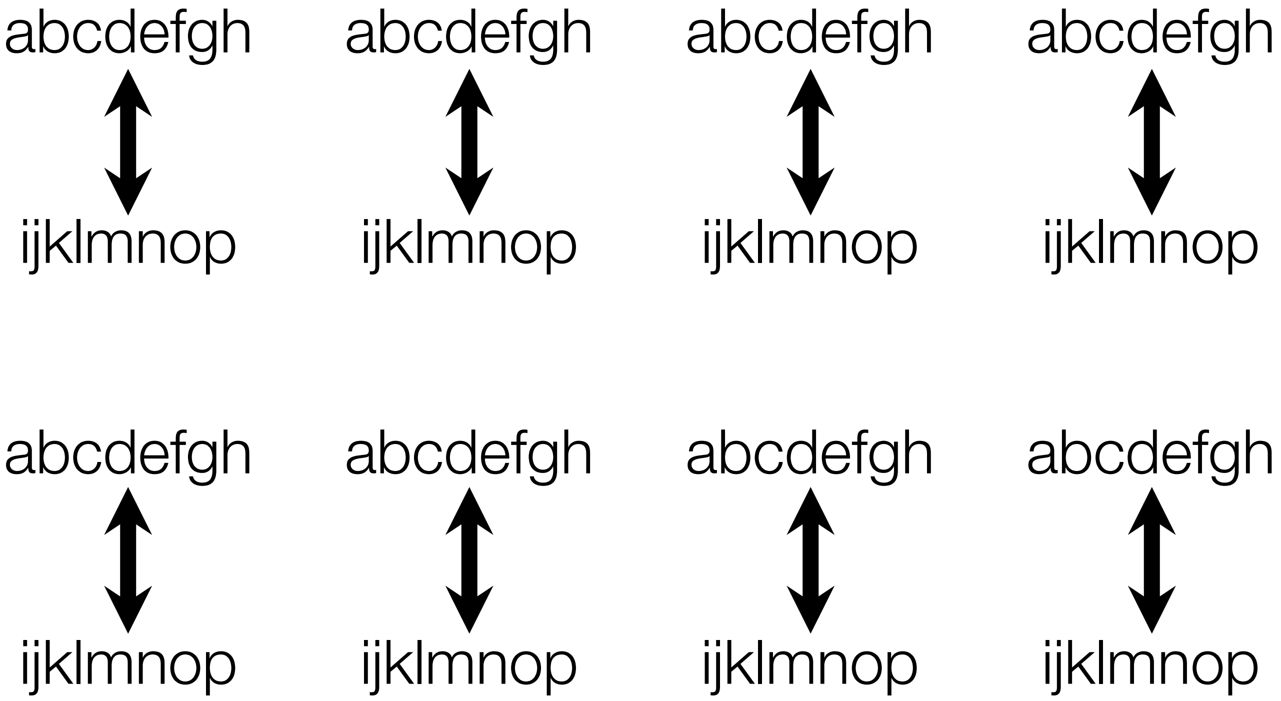
c g d h

k o l p

# Solution: Input shuffle



# Solution: Input shuffle


abcdefgh abcdefgh abcdefgh abcdefgh

ijklmnop ijklmnop ijklmnop ijklmnop

abcdefgh abcdefgh abcdefgh abcdefgh

ijklmnop ijklmnop ijklmnop ijklmnop

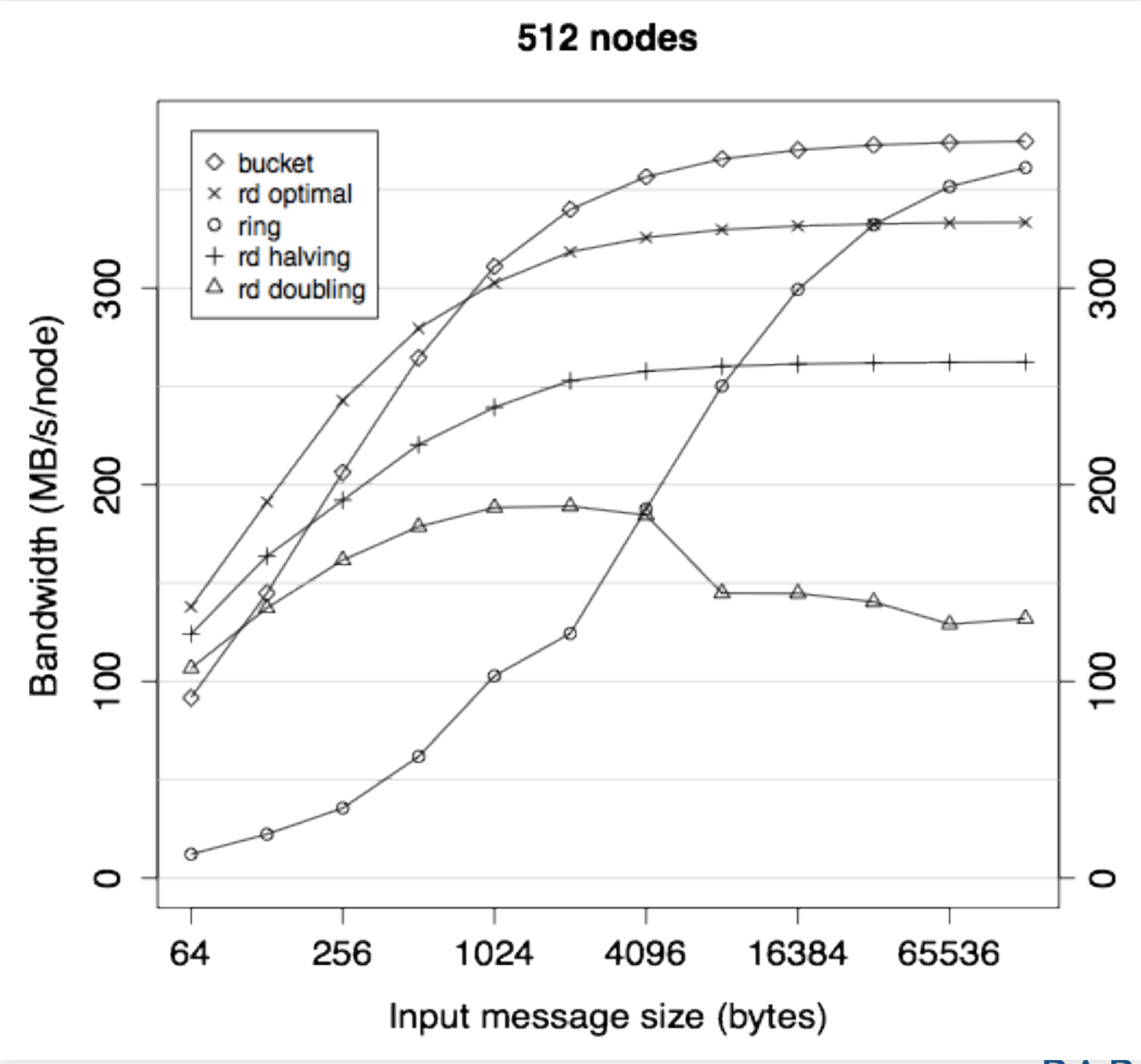
$$T=(1+lgP)\ \alpha + (7/6)nP\beta$$

$$T\approx(lgP)\alpha + (7/6)nP\beta$$

# Evaluation: Intrepid BlueGene/P at ANL

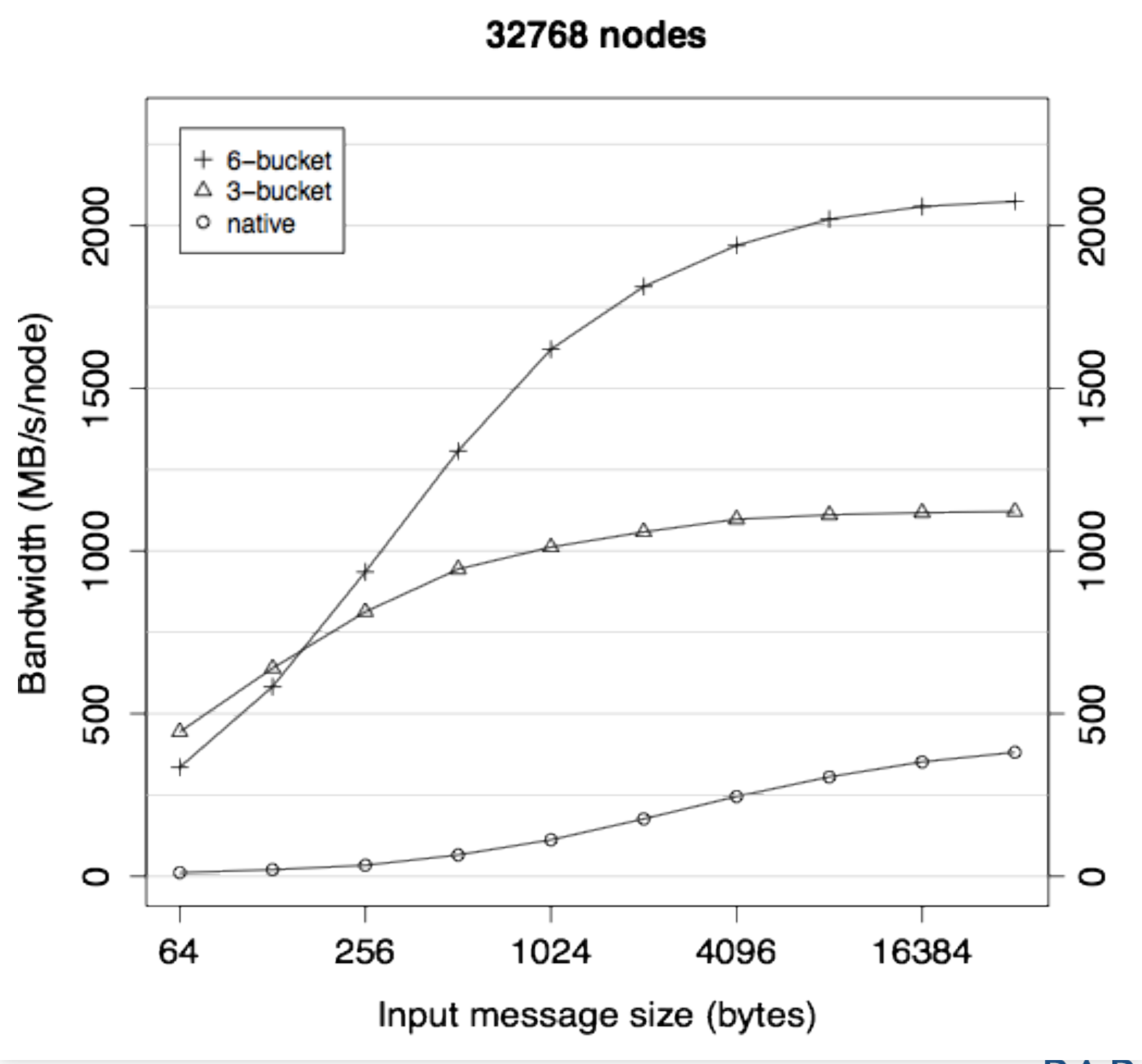
- 40k-node system
  - Each is 4 x 850 MHz PowerPC 450
- 512+ nodes is 3d torus; fewer is 3d mesh
- XLC -O4
- 375 MB/s delivered per link
  - 7% penalty using all 6 links both ways

# Allgather performance



PARALLEL@ILLINOIS

# Allgather performance



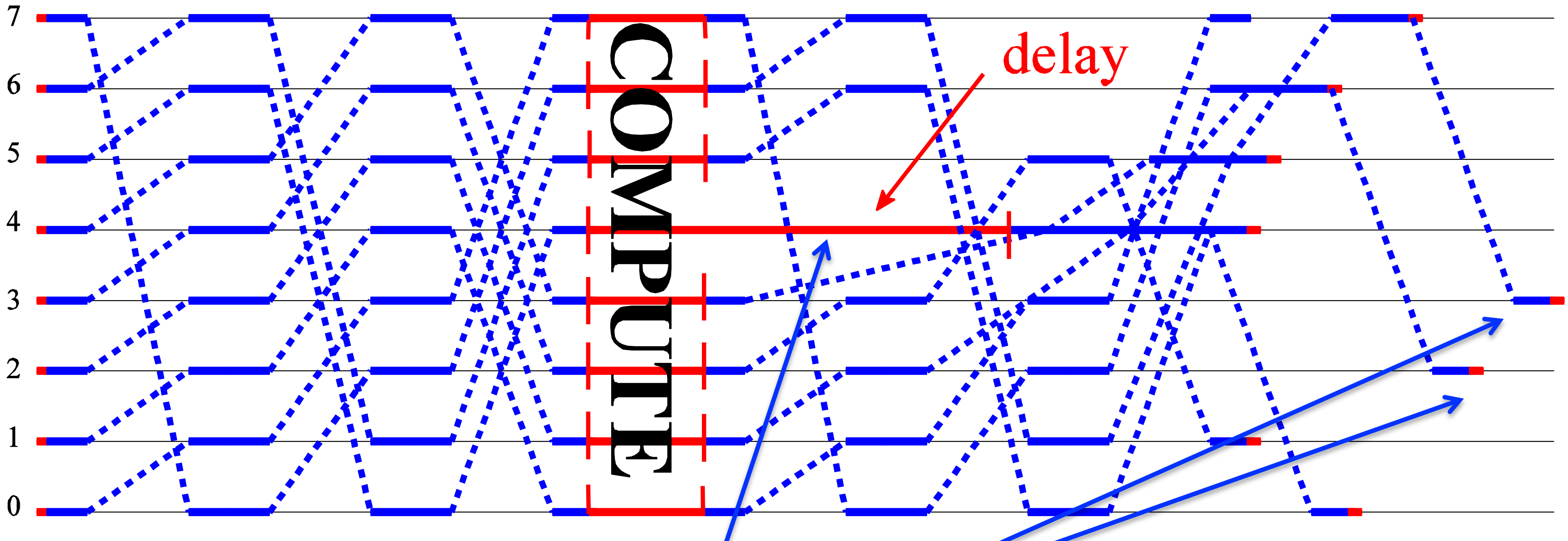
# Notes on Allgather

- 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 5 slides based on this talk…

# A Noisy Example – Dissemination Barrier



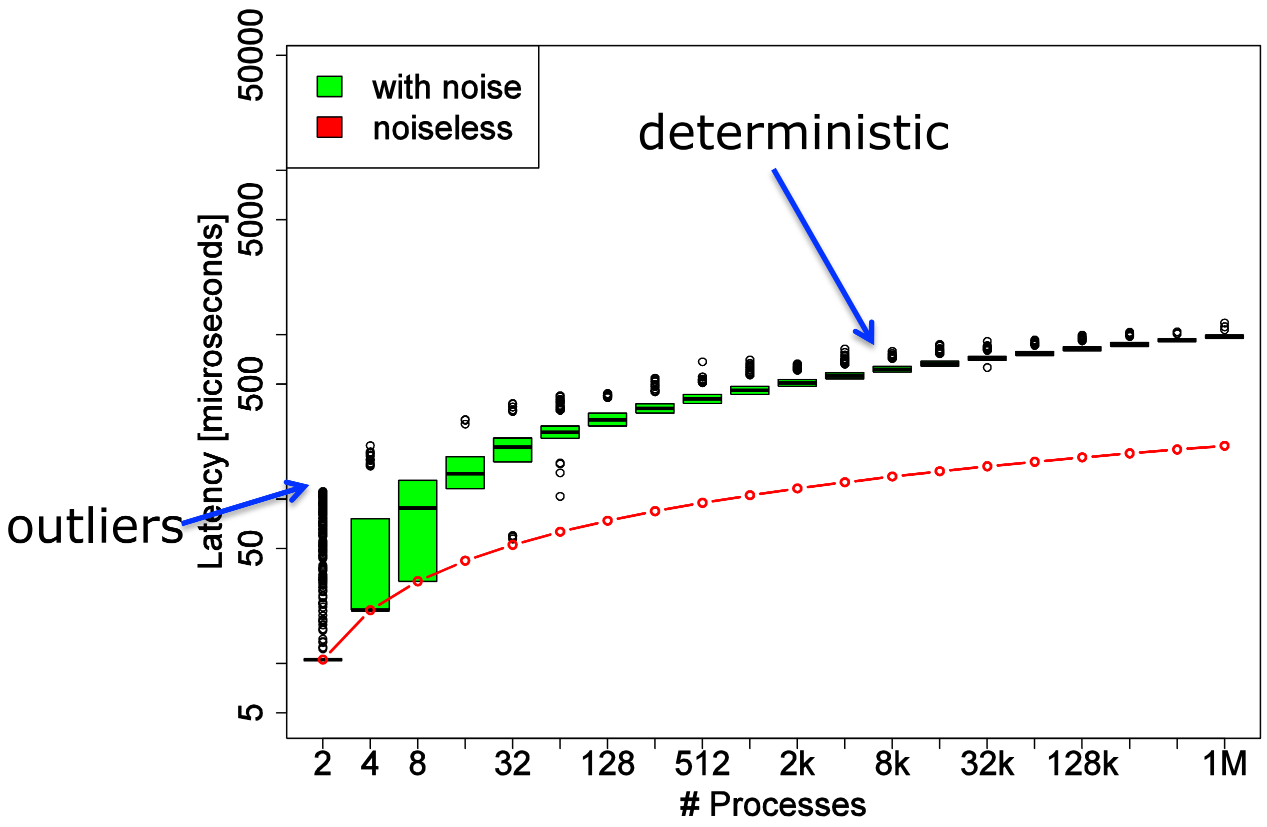
- Process 4 is delayed
  - ♦ Noise propagates "*wildly*" (of course deterministic)

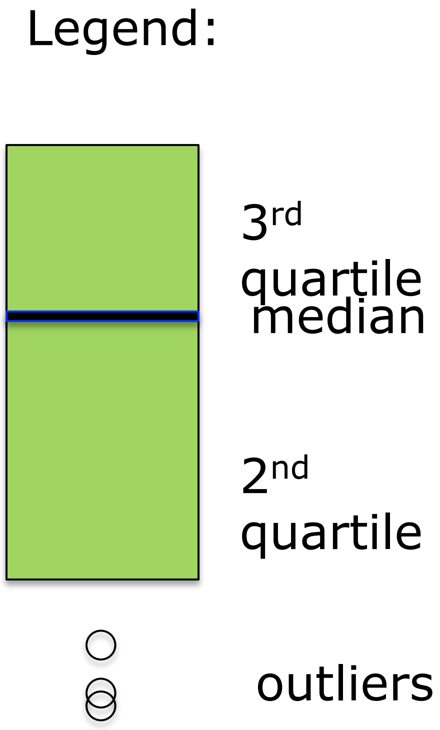
# LogGOPS Simulation Framework

- 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





- 1 Byte, Dissemination, regular noise, 1000 Hz, 100 µs

PARALLEL@ILLINOIS

# Saving Allreduce

- One common suggestion is to avoid using Allreduce
    - But algorithms with dot products are among the best known
    - Can sometimes aggregate the ate 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)
    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;
    % Can begin p'*s
    S      = M * s;
    t      = p' *s;
    alpha = gmma / t;
    x      = x + alpha * p;
    r      = r - alpha * s;
    % Can move this into the subsequent dot product
    if rnorm2 < tol2 ; break ; end
    z      = z - alpha * S;
    % Can begin r'*z here (also begin r'*r for convergence test)
    Z      = A * z;
    gmmaNew = r' * z;
    beta  = gmmaNew / gmma;
    gmma  = gmmaNew;
    % Could move x = x + alpha p here to minimize p moves.
    p      = z + beta * p;
   end
```

# Key Features

- Collective operations overlap significant local computation
- Trades extra local work for overlapped communication
  - ♦ But may not need more memory loads, so performance cost may be comparable

# Performance Analysis

- On a pure floating point basis, the nonblocking version is requires 2 more DAXPY operations
- A closer analysis shows that some operations can be merged, reducing the amount of memory motion
  - ♦ Same amount of memory motion as "classic" CG

# Processes and SMP nodes

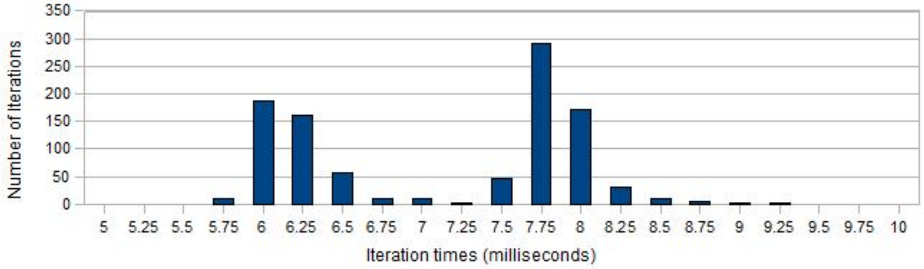
- HPC users typically believe that their code “owns” all of the cores all of the time
    - ♦ The reality is that was never true, but they did have all of the cores the same fraction of time when there was one core /node
- We can use a simple performance model to check the assertion and then use measurements to identify the problem and suggest fixes.
- Consider a simple Jacobi sweep on a regular mesh, with every core having the same amount of work.  How are run times distributed?

# Sharing an SMP

- Having many cores available makes everyone think that they can use them to solve other problems (“no one would use all of them all of the time”)
- However, compute-bound scientific calculations are often *written* as if all compute resources are owned by the application
- Such *static* scheduling leads to performance loss
- Pure dynamic scheduling adds overhead, but is better
- Careful mixed strategies are even better
- Recent results give 10-16% performance improvements on large, scalable systems
- Thanks to Vivek Kale, EuroMPI’10

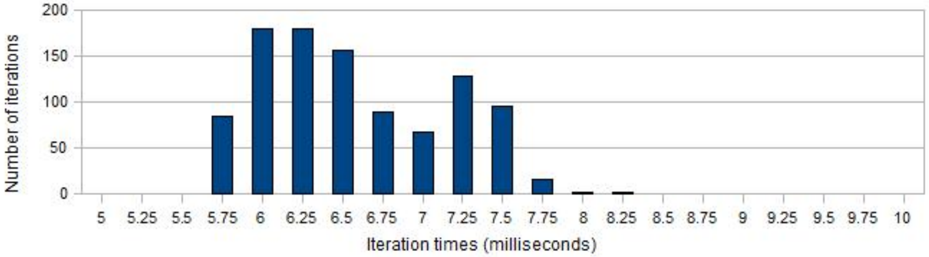
Distribution of Iteration Times for fully Static scheduling

1000 iterations , 64 x 512 x 64



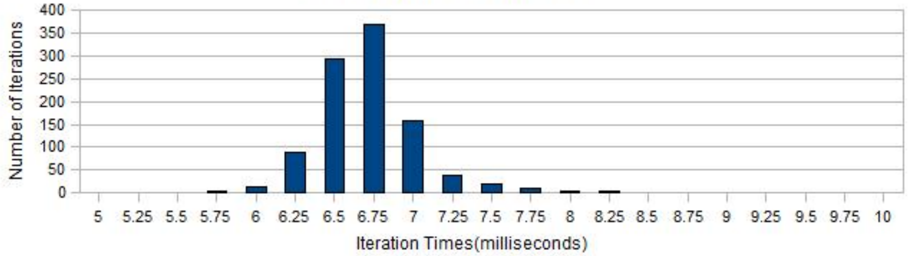
Distribution of Iteration times for 50% dynamic , with 64 tasklets

1000 iterations, 64 x 512 x 64

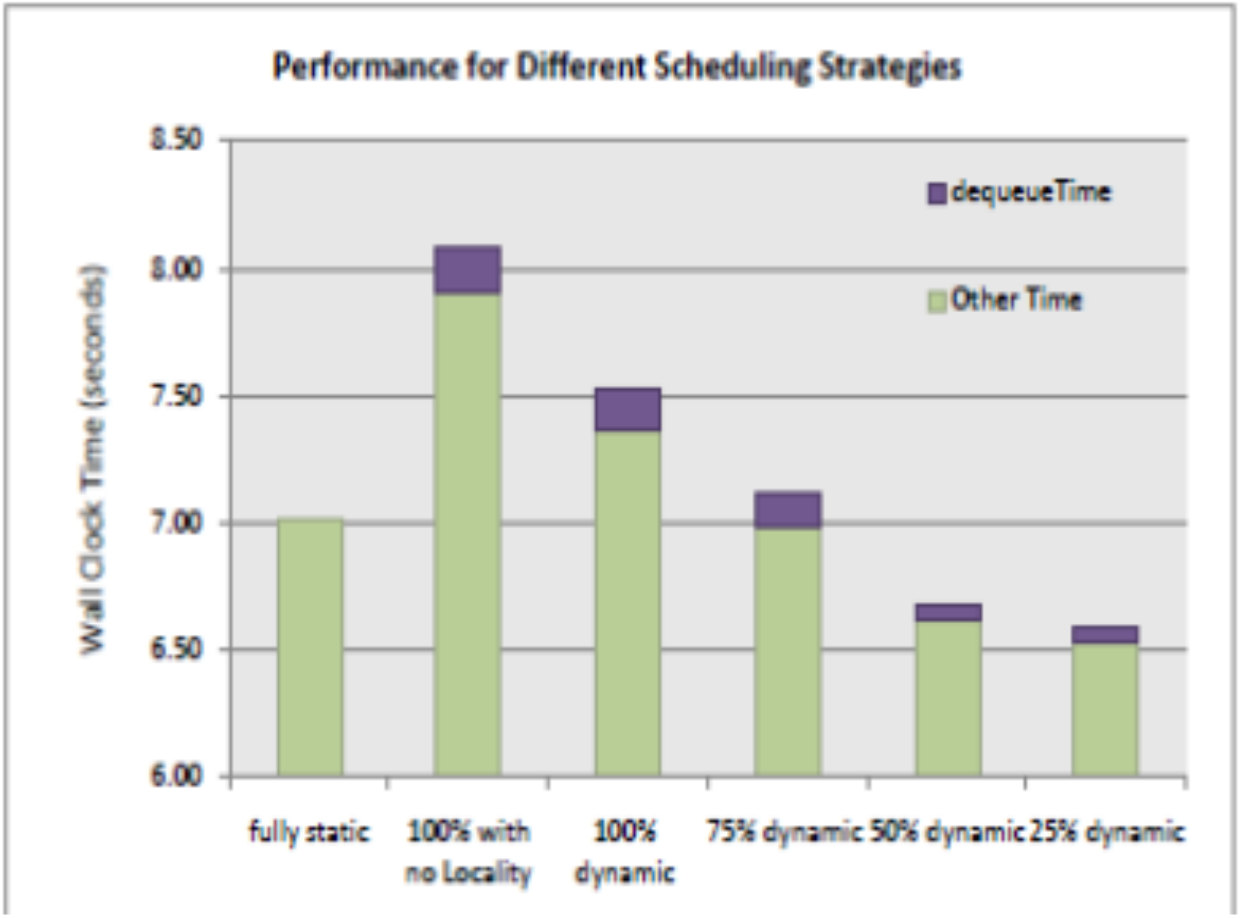


Distribution of iteration times for 50% dynamic scheduling (skewed tasklet workload)

1000 iterations, 64 x 512 x 64

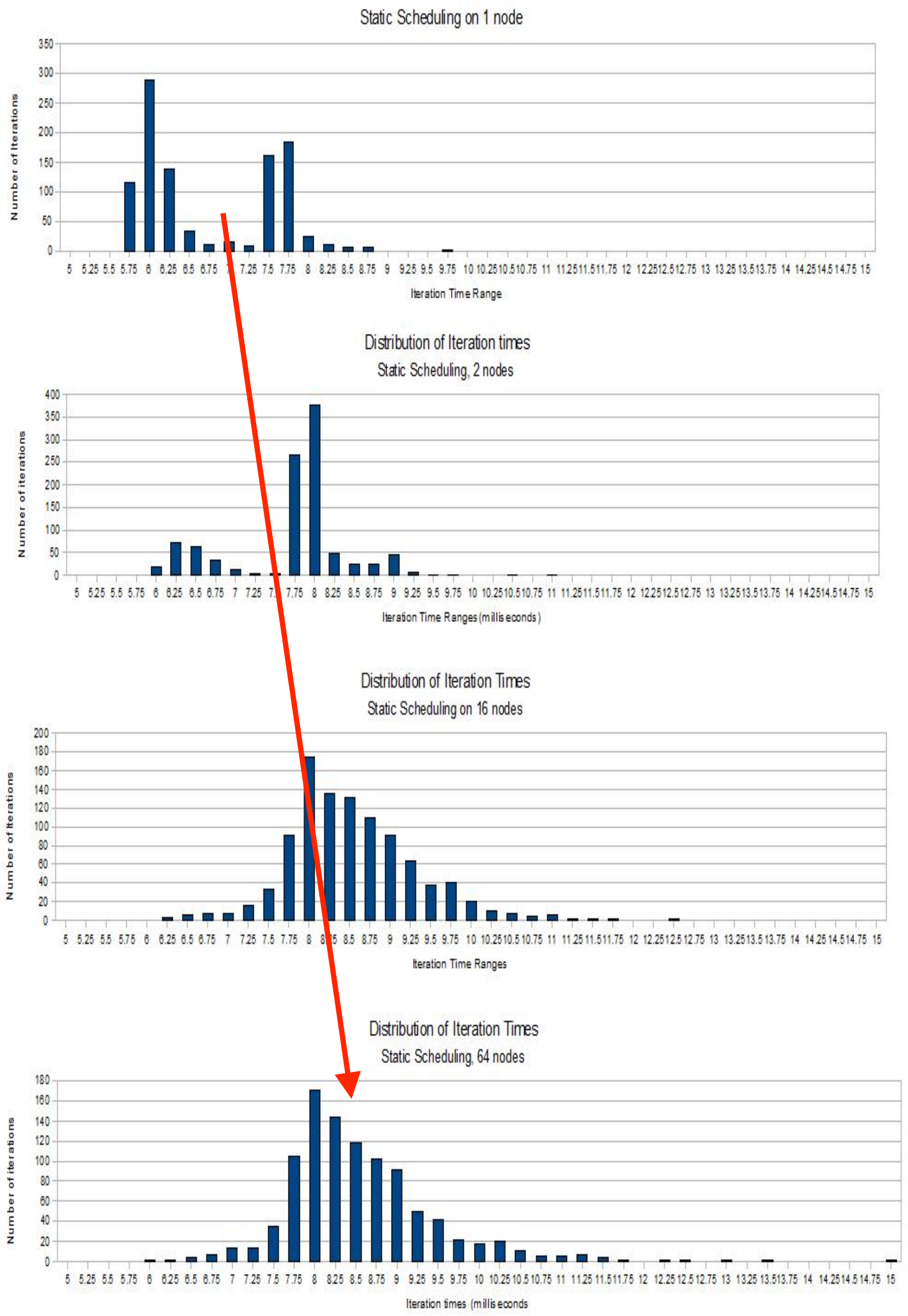


# Performance of Task Scheduling Strategies (1 node)



By doing the first portion of work using static scheduling and then doing the remainder of the work using dynamic scheduling, we achieve a consistent performance gain of nearly 8% over the traditional static scheduling method.

# Noise Amplification with an Increasing Number of Nodes



Static Scheduling on 1 node

Distribution of Iteration times

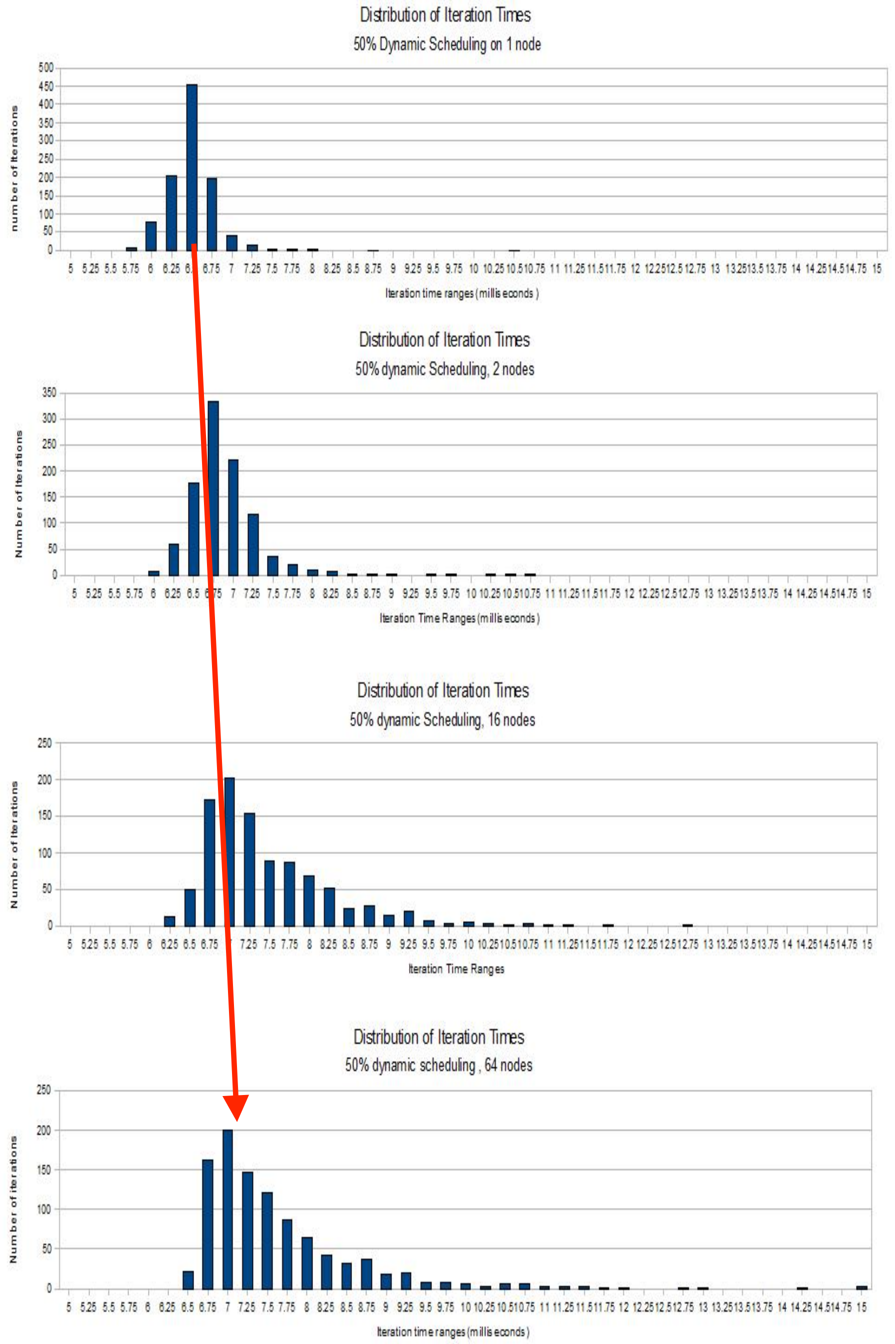
Static Scheduling, 2 nodes

Distribution of Iteration Times

Static Scheduling on 16 nodes

Distribution of Iteration Times

Static Scheduling, 64 nodes



Distribution of Iteration Times

50% Dynamic Scheduling on 1 node

Distribution of Iteration Times

50% dynamic Scheduling, 2 nodes

Distribution of Iteration Times

50% dynamic Scheduling, 16 nodes

Distribution of Iteration Times

50% dynamic scheduling , 64 nodes

# 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?

# Observations

- Details of architecture impact performance
  - ♦ Performance models can guide choices but must have enough (and only enough) detail
  - ♦ These models need only enough accuracy to guide decisions, they do not need to be predicitive
- Synchronization is the enemy
- Many techniques have been known for decades
  - ♦ We should be asking why they aren't used, and what role development environments should have

# Some Final Questions

- Is it communication avoiding or minimum solution time?
- Is it communication avoiding or latency/communication hiding?
- Is it synchronization reducing or better load balancing?
- Is it the programming model, its implementation, or its use?
- How do we answer these questions?