

Parallel Multigrid Tutorial

20th Copper Mountain Conference on
Multigrid Methods

March 26, 2021



Ulrike Meier Yang
Center for Applied Scientific Computing



References and Acknowledgements

- “Introduction to Parallel Computing”, Blaise Barney
 - https://computing.llnl.gov/tutorials/parallel_comp/
- “A Parallel Multigrid Tutorial”, Jim Jones
 - Copper Mountain tutorial in 1999, 2005, & 2007
- “A Parallel Computing Tutorial”, Ulrike Meier Yang
 - IMA Tutorial at Workshop on Fast Solution Techniques, Nov 2010
- “Parallel Multigrid Tutorial”, Rob Falgout
 - Copper Mountain tutorial in 2019

Outline

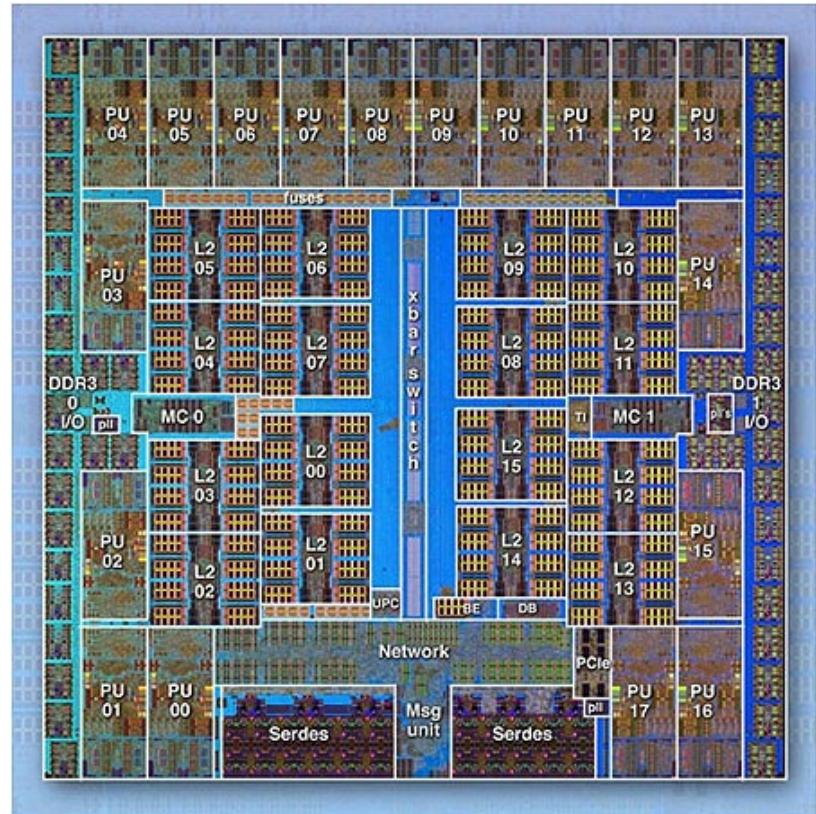
- Parallel Computing
 - Classical computer taxonomy
 - Programming models
 - Parallel performance metrics
 - Parallelizing PDE-based problems
- Parallel Multigrid
 - Parallel Algebraic Multigrid
 - Parallel Multigrid Software Design
 - Parallel in Time Multigrid

Parallel Computing



Virtually all stand-alone computers today are parallel from a hardware perspective

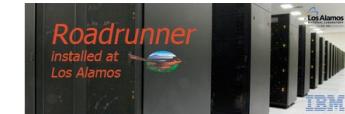
- Multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
- Multiple execution units/cores
- Multiple hardware threads



BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

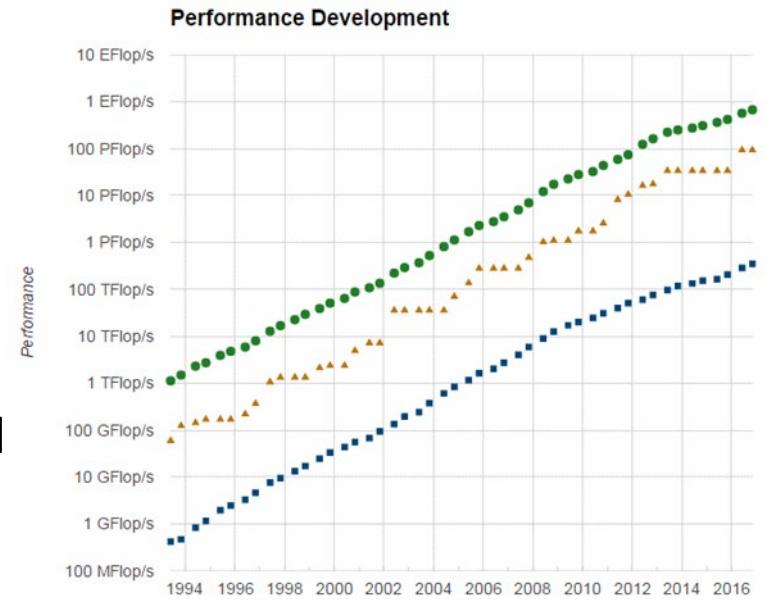
High Performance Computing

- Multigrid methods shine at large scale due to their scalability!
- New architectures impact algorithm design, requiring consideration of:
 - Parallelism:
 - Vector computers (e.g., Cray X-MP)
 - Distributed computer architectures
 - + avoiding/reducing communication and memory movement:
 - Computers with up to millions of cores (e.g., IBM Blue Gene series)
 - Hybrid computer architectures with very fast cores, complicated memory hierarchies
 - + fine-grain parallelism:
 - Heterogeneous computer architectures with accelerators/GPUs (e.g., Roadrunner, LANL, Summit, ORNL)



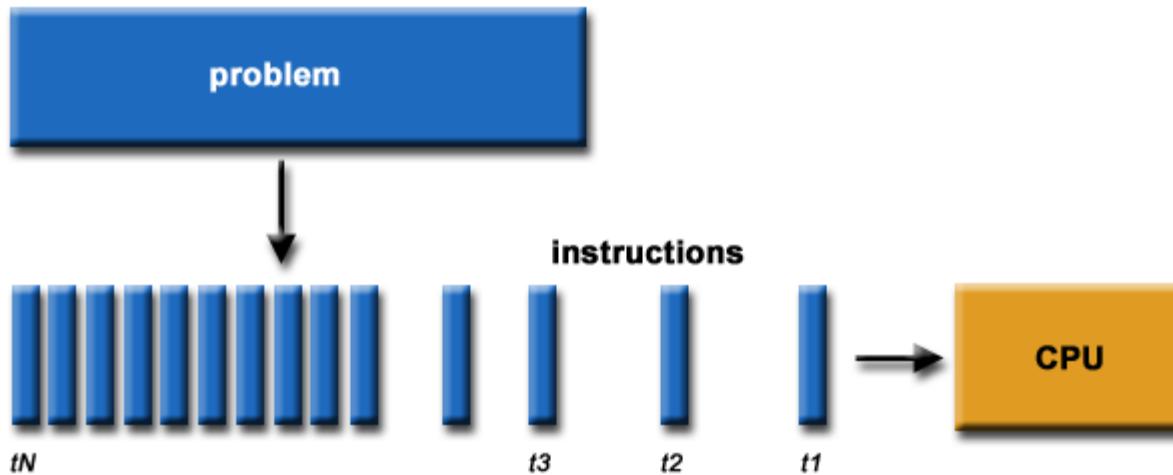
High Performance Computing – the race is on!

- Greater than 500,000x increase in supercomputer performance, with no end currently in sight!
- We are now pursuing exascale computing!
 - Exaflop = 10^{18} calculations per second
 - Fugaku first computer to achieve this using ARM chip technology
 - Other exascale systems on the horizon using GPU technology:
 - Frontier (Cray/AMD, ORNL, 2021/2022)
 - Aurora (Cray/Intel, ANL, 2021/2022)
 - El Capitan (Cray/AMD, LLNL, 2023)



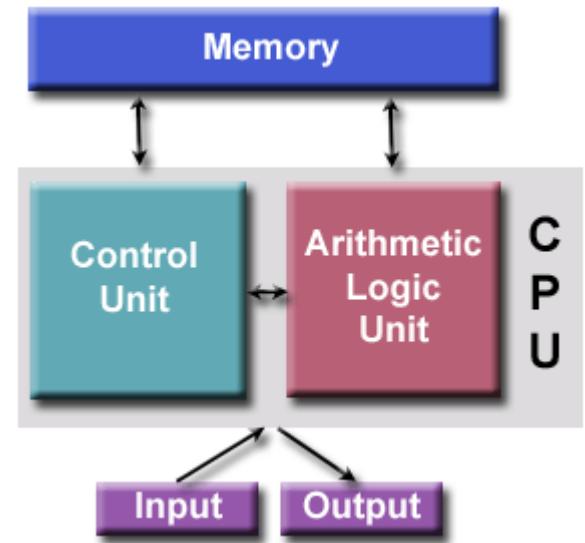
Serial computation

- Traditionally software has been written for serial computation:
 - run on a single computer
 - instructions are run one after another
 - only one instruction executed at a time



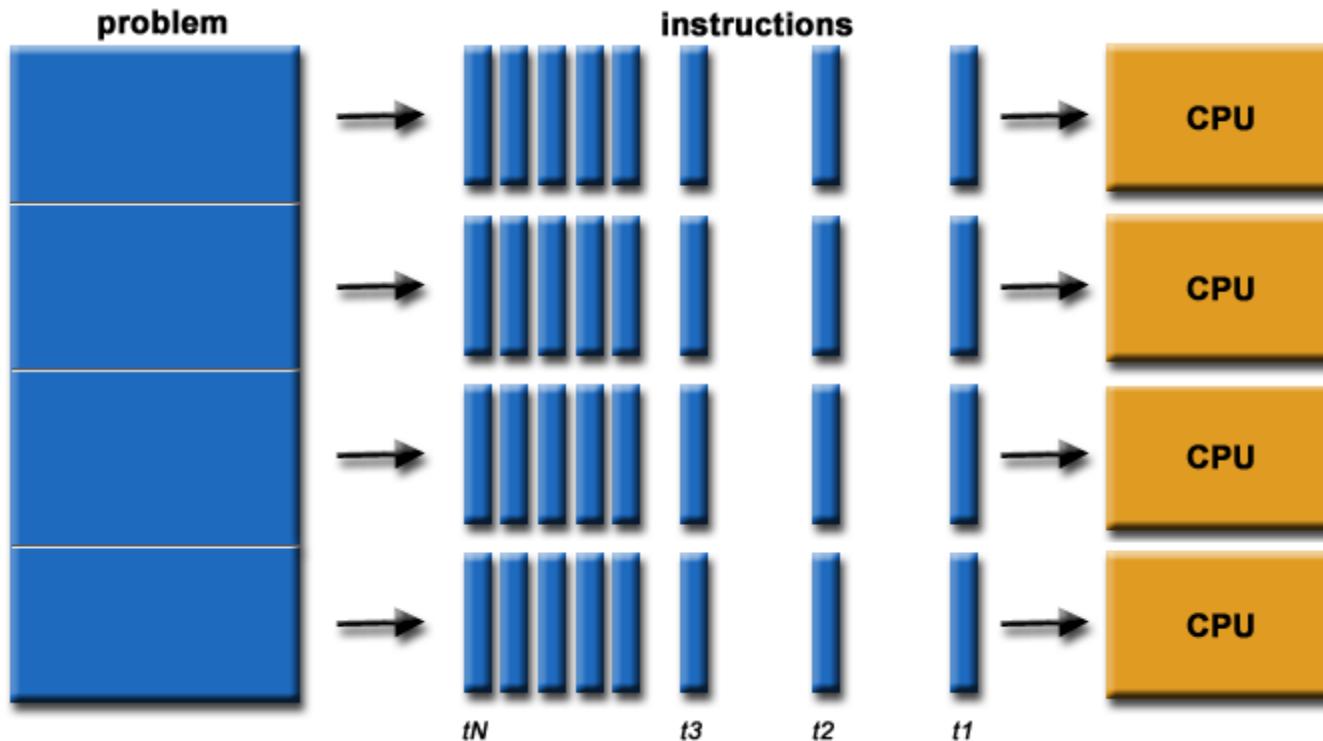
Von Neumann Architecture

- John von Neumann first authored the general requirements for an electronic computer in 1945
- Data and instructions are stored in memory
- Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.
- Arithmetic Unit performs basic arithmetic operations
- Input/Output is the interface to the human operator



Parallel Computing

- Simultaneous use of multiple compute sources to solve a single problem

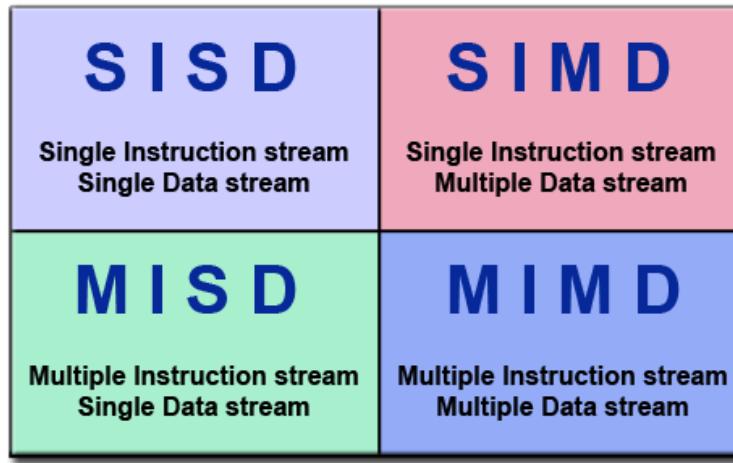


Why use parallel computing?

- Save time and/or money
- Enables the solution of larger problems
- Provide concurrency
- Use of non-local resources
- Limits to serial computing

Classical Taxonomy of Computers due to Michael Flynn in 1972

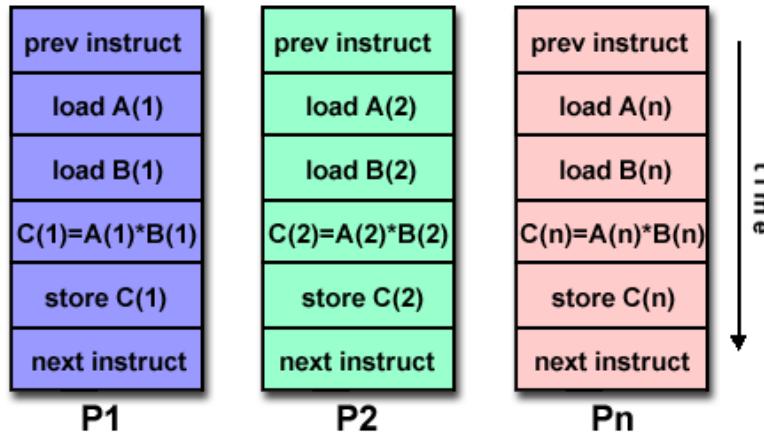
- Classifies by instruction stream and data stream



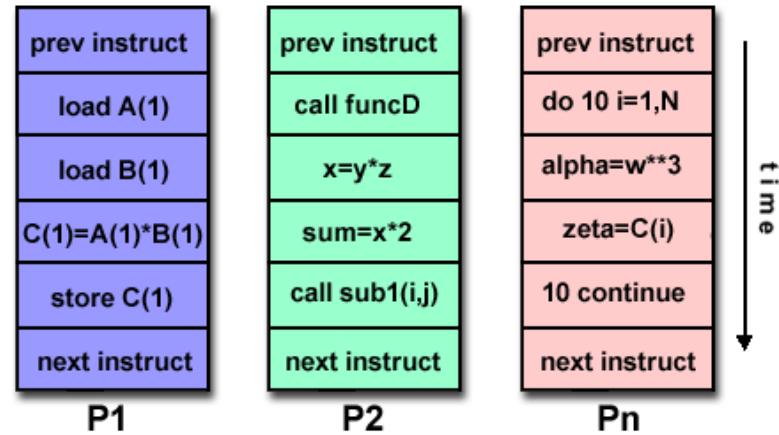
- Two main types of parallel systems today
 - SIMD: GPUs, Intel Knights Landing (KNL)
 - MIMD: Most supercomputers, clusters, multi-core PCs
- Recent supercomputers are MIMD with SIMD subcomponents

SIMD vs MIMD in Pictures

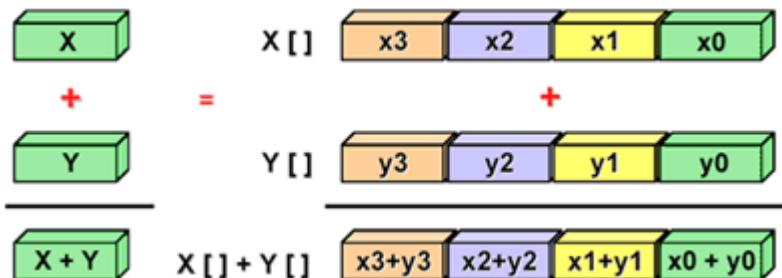
SIMD: instruction stream is the same



MIMD: instruction stream is different

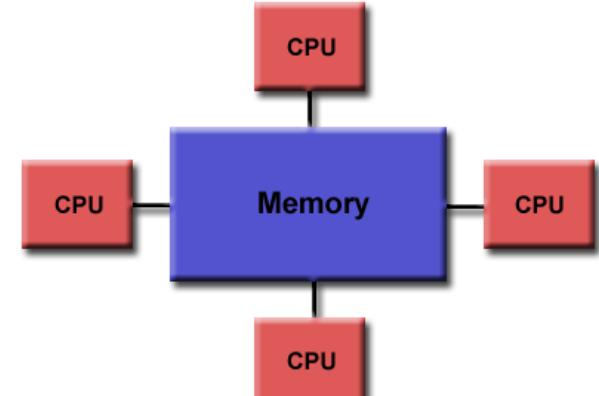


Example SIMD computation: vector addition

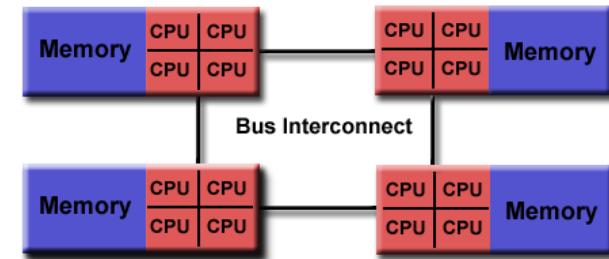


Parallel Computer Memory Architectures – Shared Memory

- All processors can access the same memory
- Uniform memory access (UMA):
 - Same access time to memory
- Non-uniform memory access (NUMA)
 - Different access times to different memories
- Advantages:
 - User-friendly programming perspective to memory (global address space)
 - Data sharing is fast
- Disadvantages:
 - Lack of scalability between memory and CPUs
 - Programmer responsible to ensure “correct” access of global memory

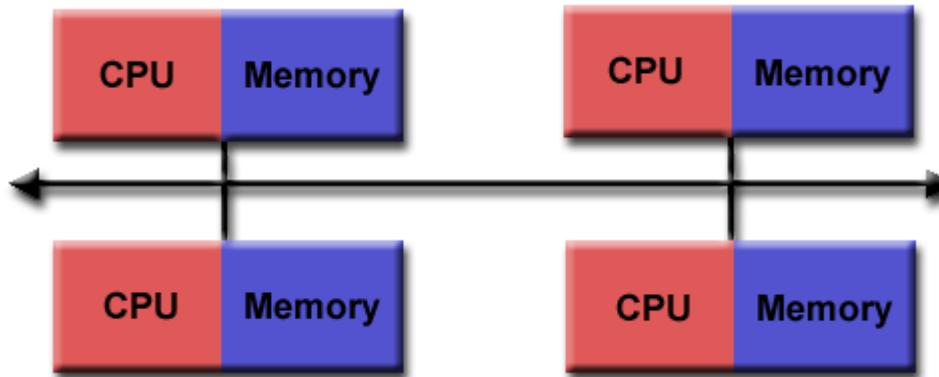


Shared Memory (UMA)



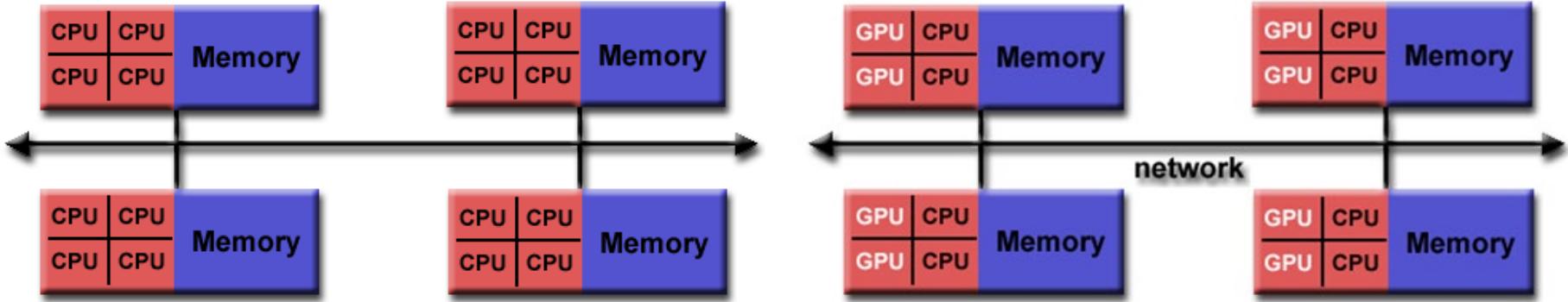
Shared Memory (NUMA)

Distributed Memory



- Distributed memory systems require a communication network to connect inter-processor memory.
- Advantages:
 - Memory is scalable with number of processors.
 - No memory interference or overhead for trying to keep cache coherency.
 - Cost effective
- Disadvantages:
 - programmer responsible for data communication between processors.
 - difficult to map existing data structures to this memory organization.

Hybrid Distributed-Shared Memory



- Generally used for the currently largest and fastest computers
- Has a mixture of previously mentioned advantages and disadvantages

Parallel Programming Models

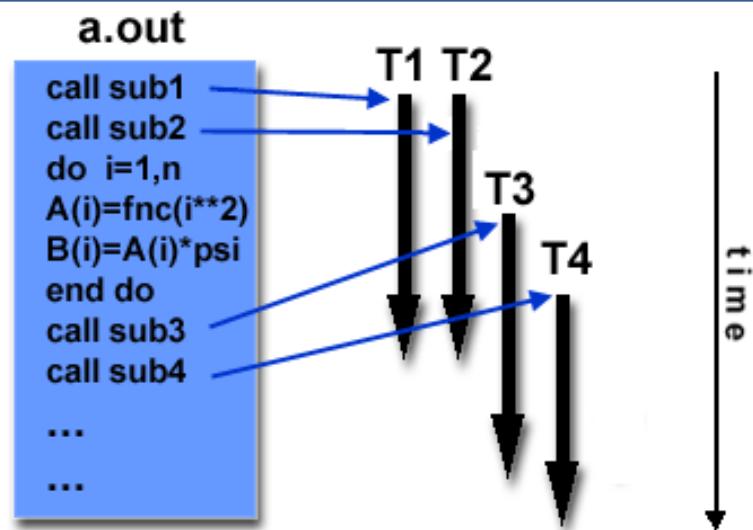
- Shared memory (without threads)
 - API in many operating systems, SHMEM
- Shared memory with threads
 - POSIX Threads (Pthreads), Intel TBB, OpenMP (since 1997)
- Message Passing
 - MPI (Message Passing Interface, since 1994)
- Data Parallel
 - Also referred to as Partitioned Global Address Space (PGAS) model
 - Coarray Fortran, Unified Parallel C (UPC), Global Arrays, X10, Chapel
- GPU/accelerator
 - CUDA (Nvidia GPUs only)
 - HIP (Nvidia, AMD)
 - DPC++/SYCL (target more general GPUs)
- Hybrid
 - MPI+OpenMP, MPI+CUDA, etc.
- **All of these can be implemented on any architecture**
 - Most common approach – Single Program Multiple Data (SPMD)

Shared Memory

- Tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- Advantage: no need to explicitly communicate data between tasks -> simplified programming
- Disadvantages:
 - Need to take care when managing memory, avoid synchronization conflicts, race conditions
 - Harder to control data locality

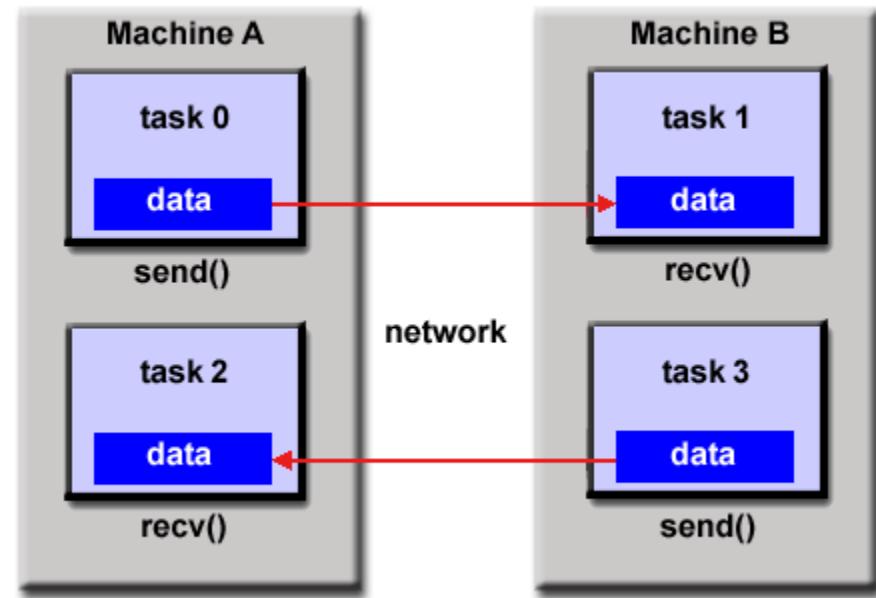
Threads

- A thread can be considered as a subroutine in the main program
- Threads communicate with each other through the global memory
- commonly associated with shared memory architectures and operating systems
- Posix Threads or pthreads
- OpenMP



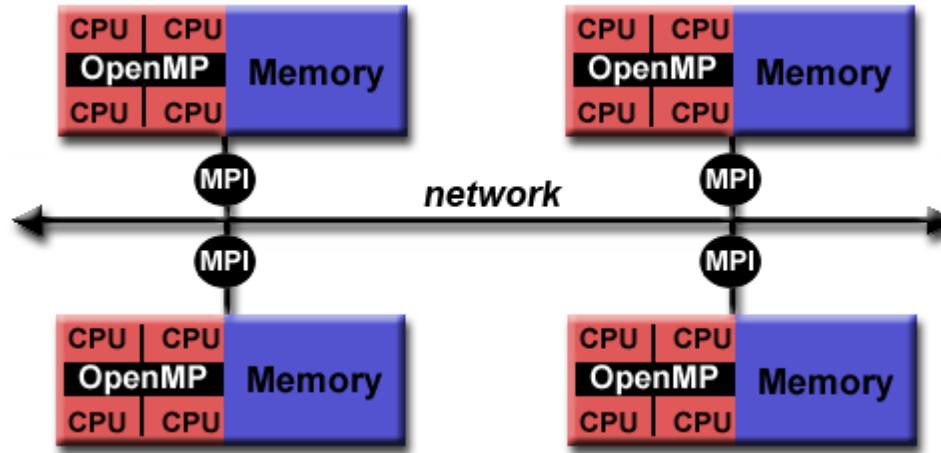
Message Passing

- A set of tasks that use their own local memory during computation.
- Data exchange through sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.
- MPI (released in 1994)
- MPI-2 (released in 1996)
- MPI-3 (released in 2012)
- MPI-4 (in progress)



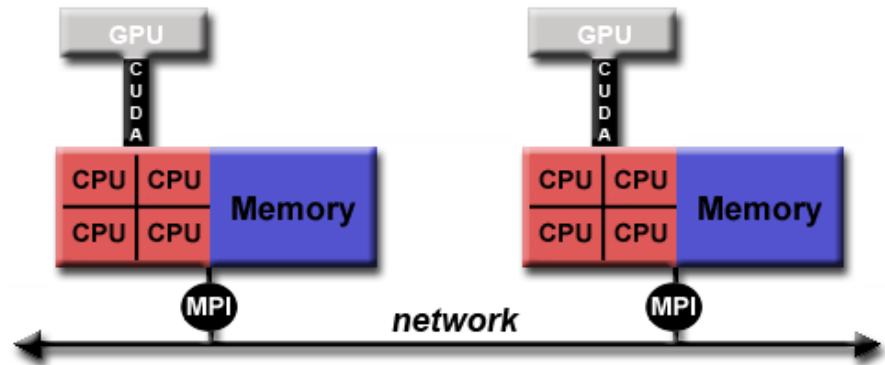
Hybrid Programming Models

- Combines more than one programming model.
- MPI+OpenMP
 - Threads perform computationally intensive kernels using local, on-node data
 - Communications between processes on different nodes occurs over the network using MPI



Hybrid Programming Models (2)

- MPI with CPU-GPU programming.
 - MPI tasks run on CPUs using local memory and communicate over network.
 - Computationally intensive kernels off-loaded to GPUs on-node.
 - Data exchange between node-local memory and GPUs uses CUDA (or something equivalent).
 - Note threads (OpenMP) can be used on CPUs in addition to MPI
 - Ability to use unified virtual memory (UVM) as memory shared by GPUs and CPUs; facilitates programming, but often performs worse.



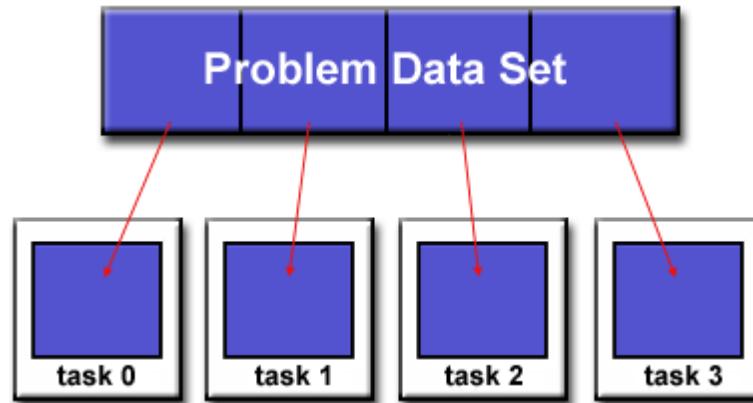
This tutorial will discuss parallel multigrid from a distributed-memory message-passing viewpoint

- Will mostly use conceptual diagrams and descriptions
- Will not focus on implementation details and specific code
- MPI and OpenMP will be the primary languages mentioned



Designing Parallel Programs

- Examine problem:
 - Can the problem be parallelized?
 - Are there data dependencies?
 - where is most of the work done?
 - identify bottlenecks (e.g., I/O)
- Partitioning
 - How should the data be decomposed?



Communication – Two basic types

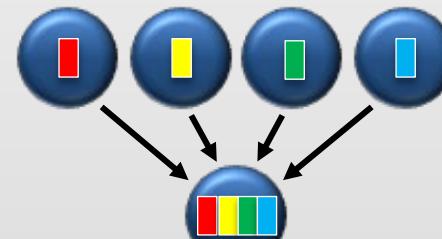
Point-to-point



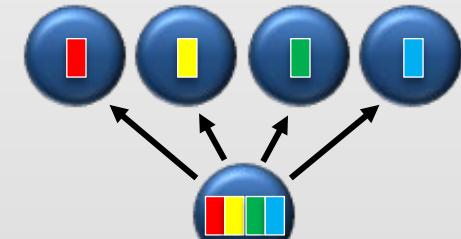
Send

Receive

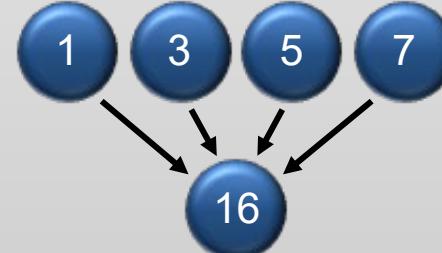
Collective



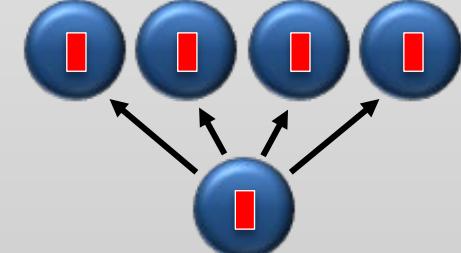
Gather



Scatter



Reduction



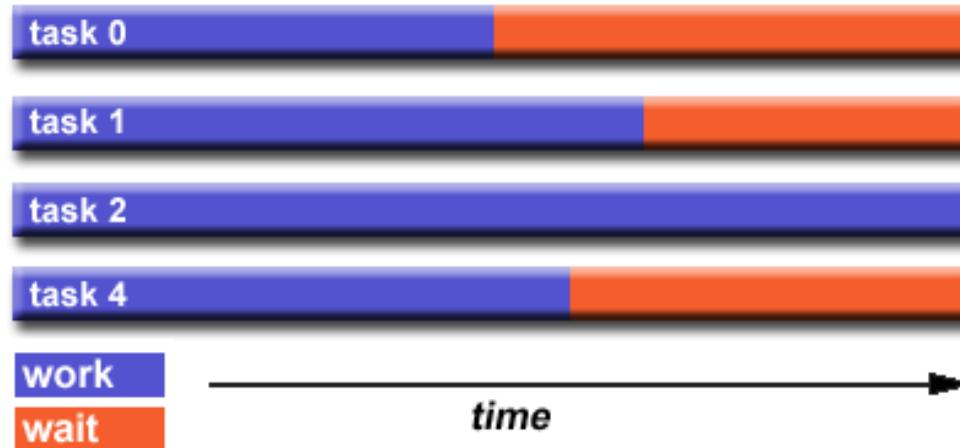
Broadcast

Allreduce \equiv Reduction + Broadcast

- MG codes mainly use
 - Send, Receive
 - Allreduce (MPI routine)
- Gather and Scatter are not scalable

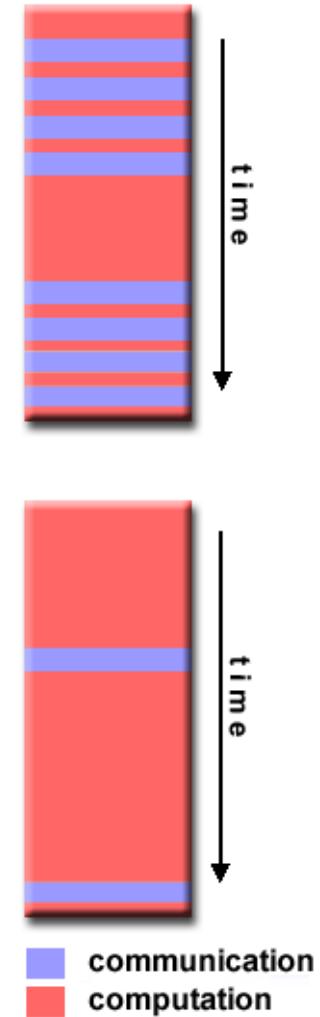
Load balancing

- Keep all tasks busy all of the time
 - Minimize idle time
- The slowest task will determine the overall performance



Granularity

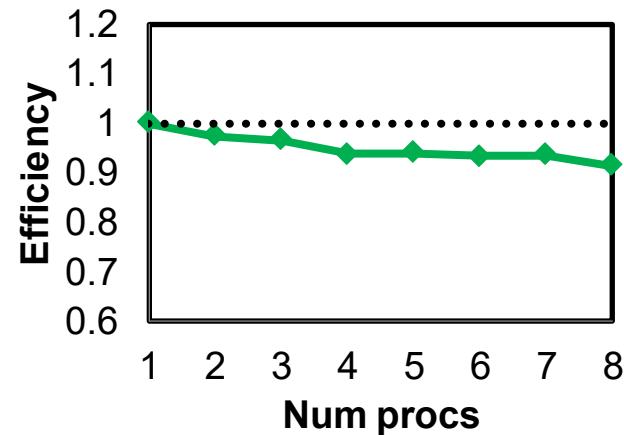
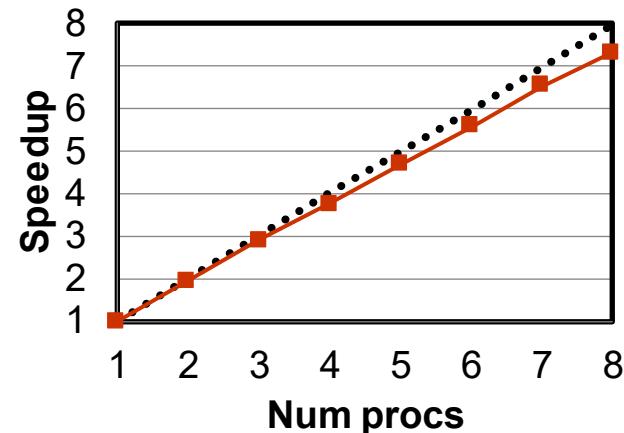
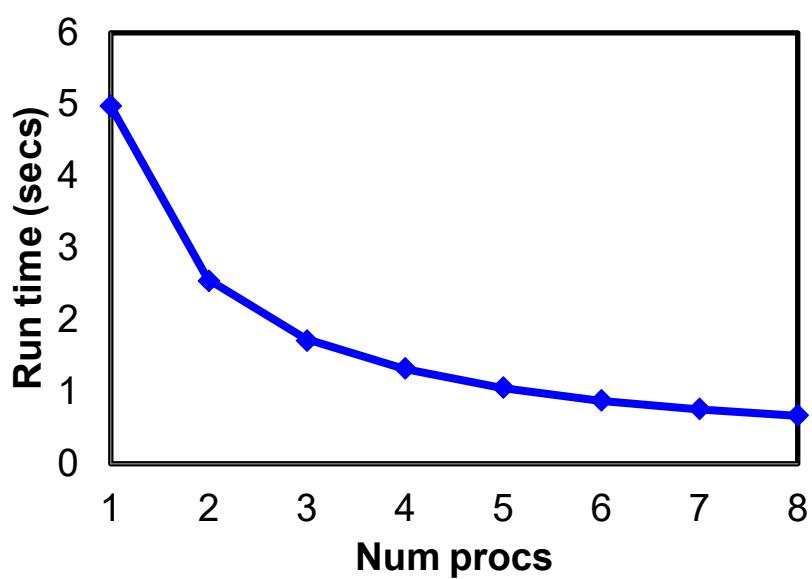
- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- Fine-grain Parallelism:
 - Low computation to communication ratio
 - Facilitates load balancing
 - Implies high communication overhead and less opportunity for performance enhancement
 - needed for good GPU performance!
- Coarse-grain Parallelism:
 - High computation to communication ratio
 - Implies more opportunity for performance increase
 - Harder to load balance efficiently



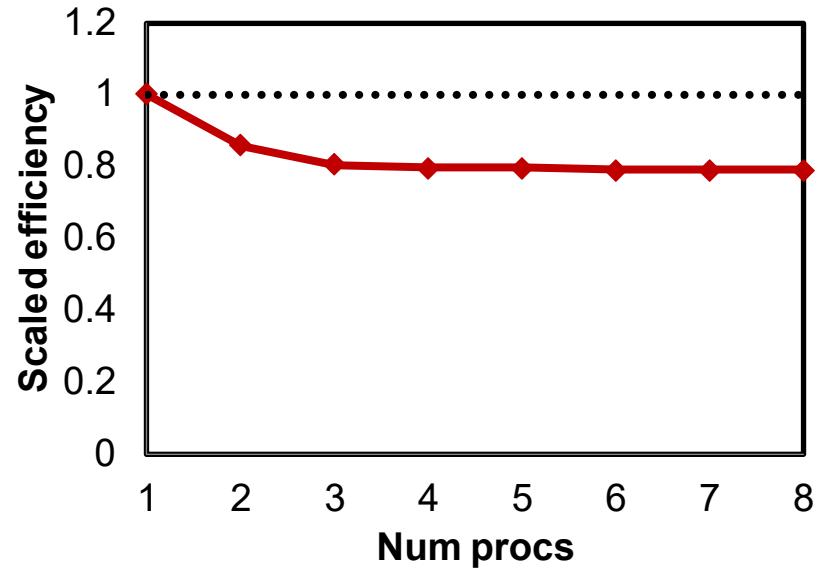
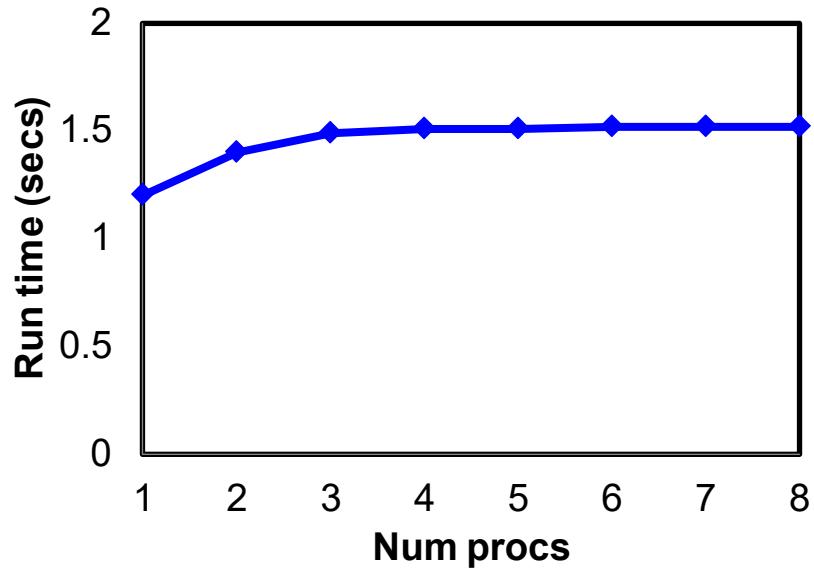
Parallel Computing Performance Metrics

- Let $T(N, P)$ be the time to solve a problem of size N using P processors
- **Strong scaling** – problem size is fixed
 - Speedup: $S(N, P) = T(N, 1) / T(N, P)$
 - Efficiency: $E(N, P) = S(N, P) / P$
- **Weak scaling** – problem size is proportional to P
 - Scaled Efficiency: $SE(N, P) = T(N, 1) / T(PN, P)$
- An algorithm is **scalable** if $T(PN, P) \leq C$ for all P
 - Ideally C is a constant, but this is not always possible
 - For multigrid, C grows as $\log P$ (this is optimal for some problems)

Strong Scaling – fix the problem size and increase the number of processors



Weak Scaling – increase the problem size and the number of processors proportionately

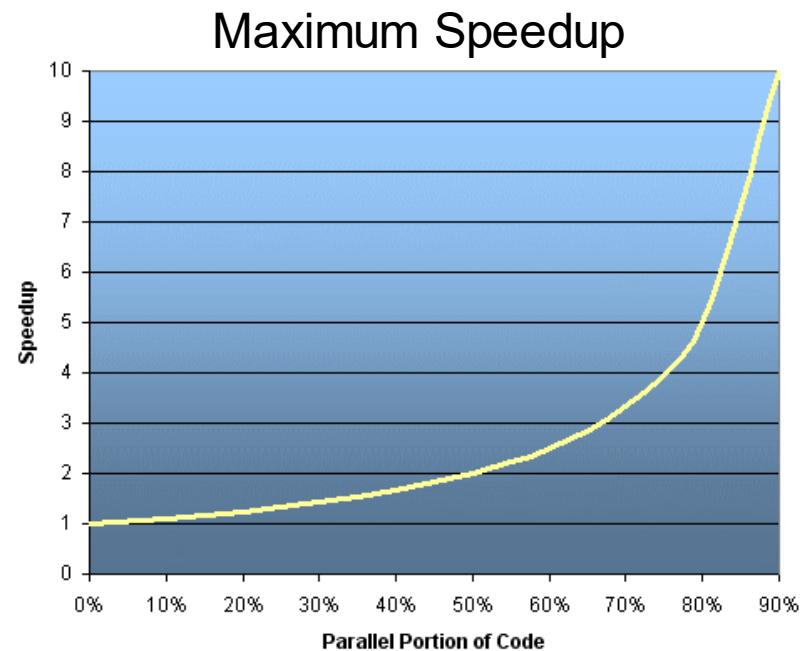
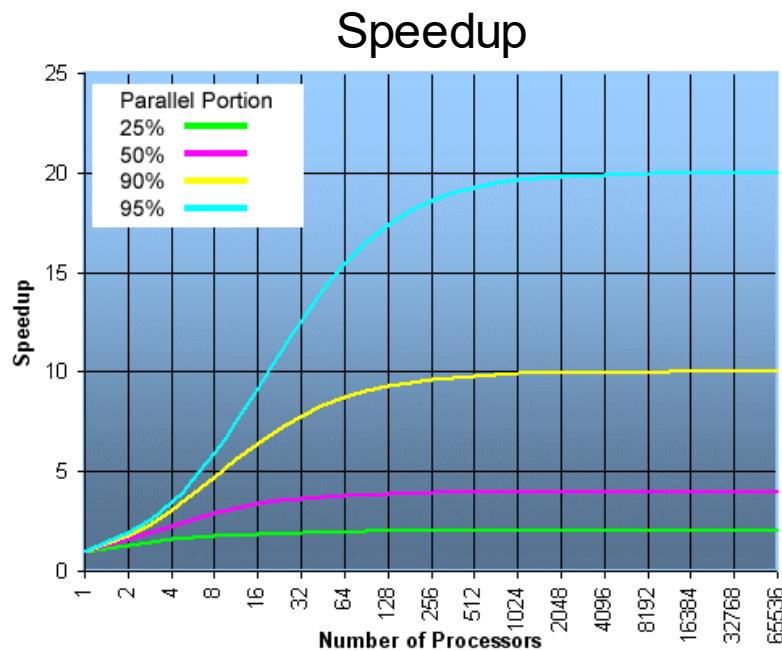


Amdahl's Law models speedup as a function of the serial (non-parallelizable) component of a code

$$\text{Speedup} = \frac{1}{F/P + (1 - F)} < \frac{1}{(1 - F)}$$

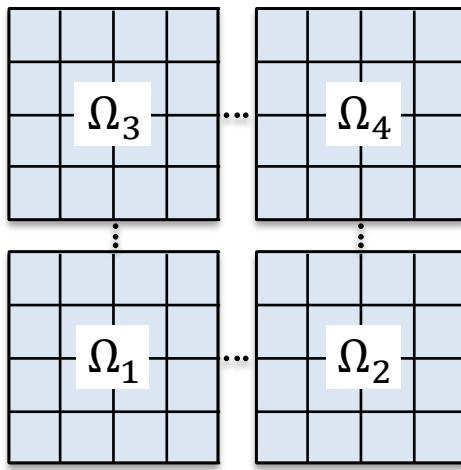
Parallelizable fraction Num processors Serial fraction

Maximum Speedup



Domain partitioning is the primary approach for parallelizing PDE-based problems

- Example grid Ω partitioned into 4 subdomains Ω_p



- Example sparse linear system resulting from a discretized PDE (e.g., 5-pt cell-centered discretization of $\Delta u = f$)

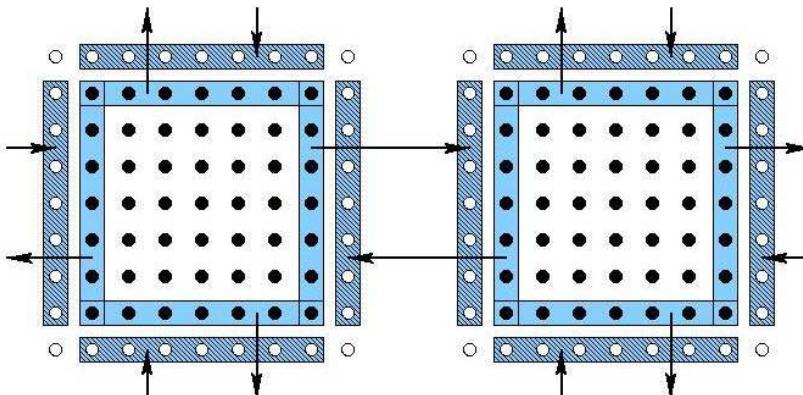
$$\begin{matrix} & \vdots & & \vdots & \\ & A_1 & & & u_1 \\ & \vdots & & \vdots & \\ A_2 & & & & u_2 \\ & \vdots & & \vdots & \\ A_3 & & & & u_3 \\ & \vdots & & \vdots & \\ A_4 & & & & u_4 \end{matrix} = \begin{matrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{matrix}$$

- Processor p owns data associated with Ω_p

- Each 5-pt stencil corresponds to a row of A
- Processor p owns A_p, u_p, f_p

Basic linear algebra operations are at the core of parallel multigrid algorithms

- Matrix-vector multiply: Ax
- On each processor p :
 - Exchange subdomain boundary data with “nearest neighbors” (MPI Send/Recv)
 - Compute local product $A_p x_p$



- Vector addition: $z = x + y$

$$\begin{array}{rcl} z_0 & = & x_0 + y_0 \leftarrow \text{Proc 0} \\ z_1 & = & x_1 + y_1 \leftarrow \text{Proc 1} \\ z_2 & = & x_2 + y_2 \leftarrow \text{Proc 2} \end{array}$$

- Vector product: $s = x^T y$
 - Compute local product then $s_0 + s_1 + s_2$ (MPI Allreduce)

$$\begin{array}{rcl} s_0 & = & x_0 * y_0 \leftarrow \text{Proc 0} \\ s_1 & = & x_1 * y_1 \leftarrow \text{Proc 1} \\ s_2 & = & x_2 * y_2 \leftarrow \text{Proc 2} \end{array}$$

Parallel programming models provide useful insight into expected performance

- Most common communication/computation model
 - Simple, but effective for providing qualitative understanding
 - Better models account for message contention, network topology, etc.

$$T_{comm} = \alpha + m\beta \quad (\text{communicate } m \text{ doubles})$$

$$T_{comp} = m\gamma \quad (\text{compute } m \text{ flops})$$

- Values for α, β, γ vary across machines, but communication generally dominates, especially network latency

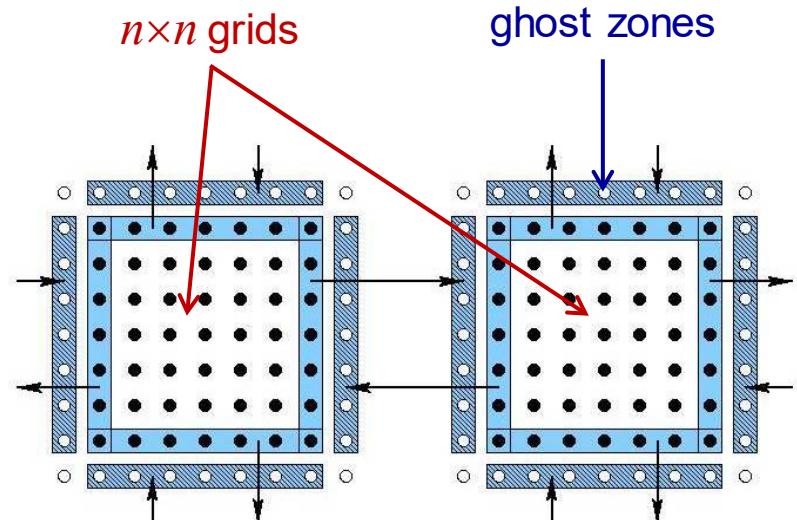
Typical relationship: $\alpha = 10^4\gamma; \beta = 10^1\gamma$

- For sparse linear algebra (especially multigrid), developing approaches to **minimize communication** is key

Parallel model for matrix-vector multiply – 5-pt discretization of 2D Laplace equation

- On each processor p :
 - Exchange subdomain boundary data with “nearest neighbors”
 - Compute local product $A_p x_p$
- Total time determined by slowest processor
- Time to do a matvec
 - 4 communications of size n data (assuming bi-directional)
 - $5n^2$ computations (multiply-adds)

$$T \approx 4\alpha + 4n\beta + 5n^2\gamma$$

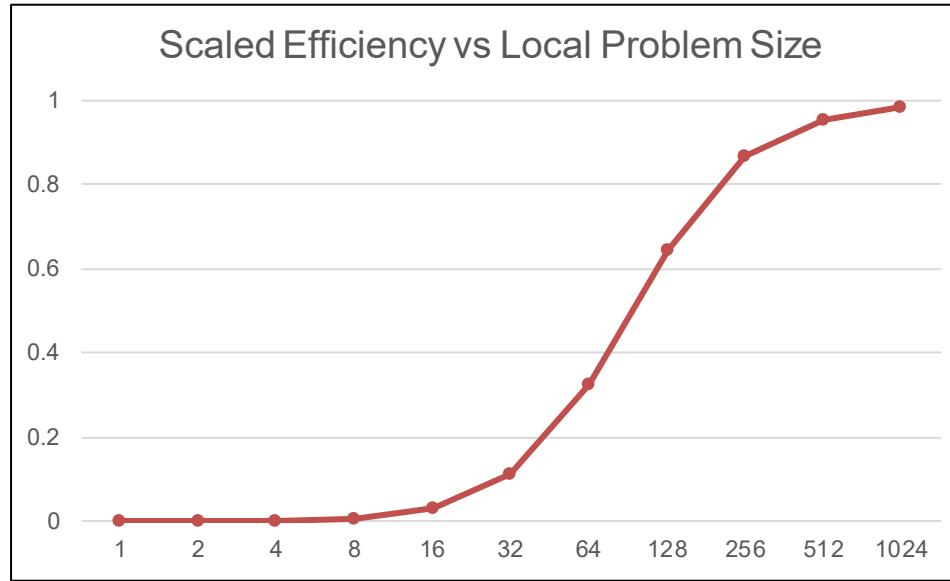


5-pt stencil

$$\begin{bmatrix} -1 & & -1 \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

Increase local problem size for efficiency

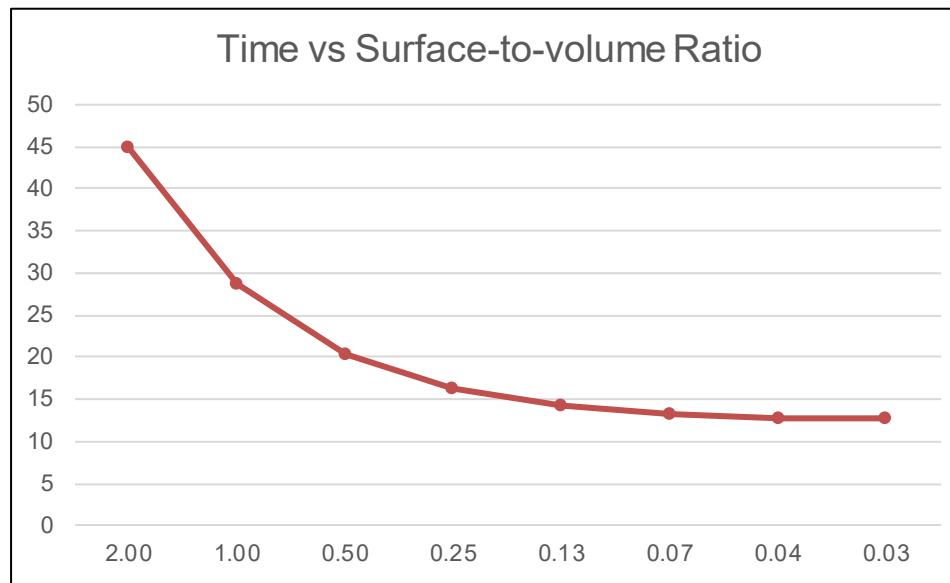
- Large local problem size → computations dominate



- It's not always possible to increase problem size
 - Competition for memory with the rest of the application
- Easy to make an algorithm look good by choosing large problem
 - Better to show both large and small cases

Minimize surface-to-volume ratio for speed

- Consider a local volume of size 16,384 ($=128^2$), but for rectangles of varying dimensions
 - Large surface-to-volume ratio = long thin rectangle ($1 \times 16,384$)
 - Smallest ratio = a cube (128×128)

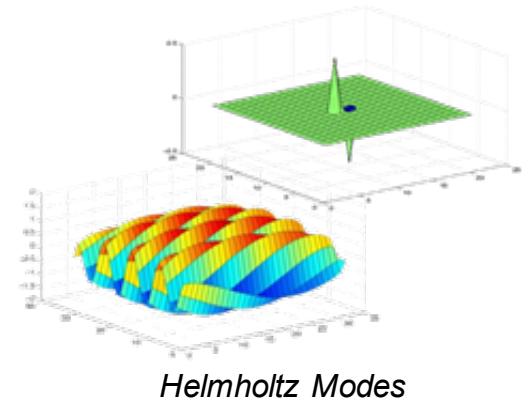
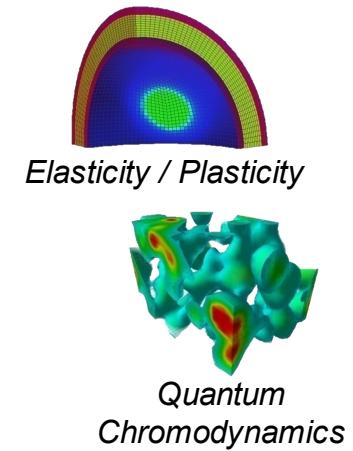


Parallel Multigrid

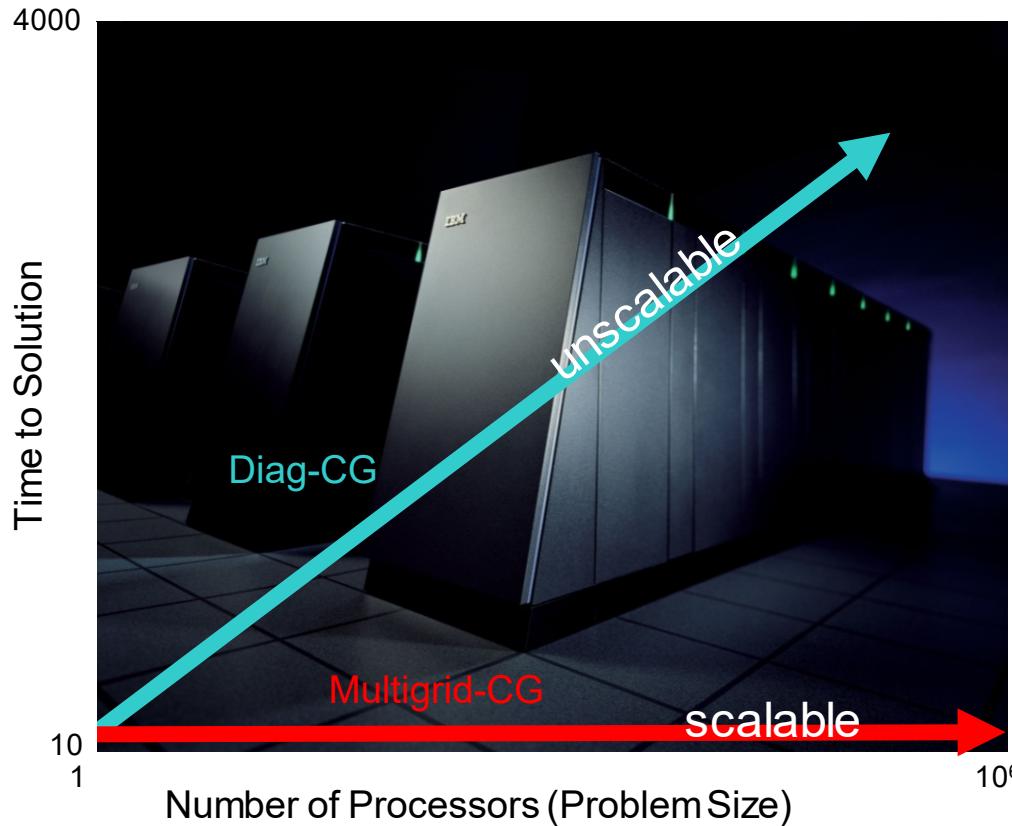


Multigrid is playing an important role for addressing exascale challenges

- For many applications, the fastest and most scalable solvers are multigrid methods
- Exascale solver algorithms need to:
 - Exhibit extreme levels of parallelism (**exascale → 1B cores**)
 - Minimize data movement & exploit machine heterogeneity
 - Demonstrate resilience to faults
- **Multilevel methods are ideal**
 - Key feature: Optimal $O(N)$
- Research challenge:
 - No optimal solvers yet for some applications, even in serial!
 - Parallel computing increases difficulty

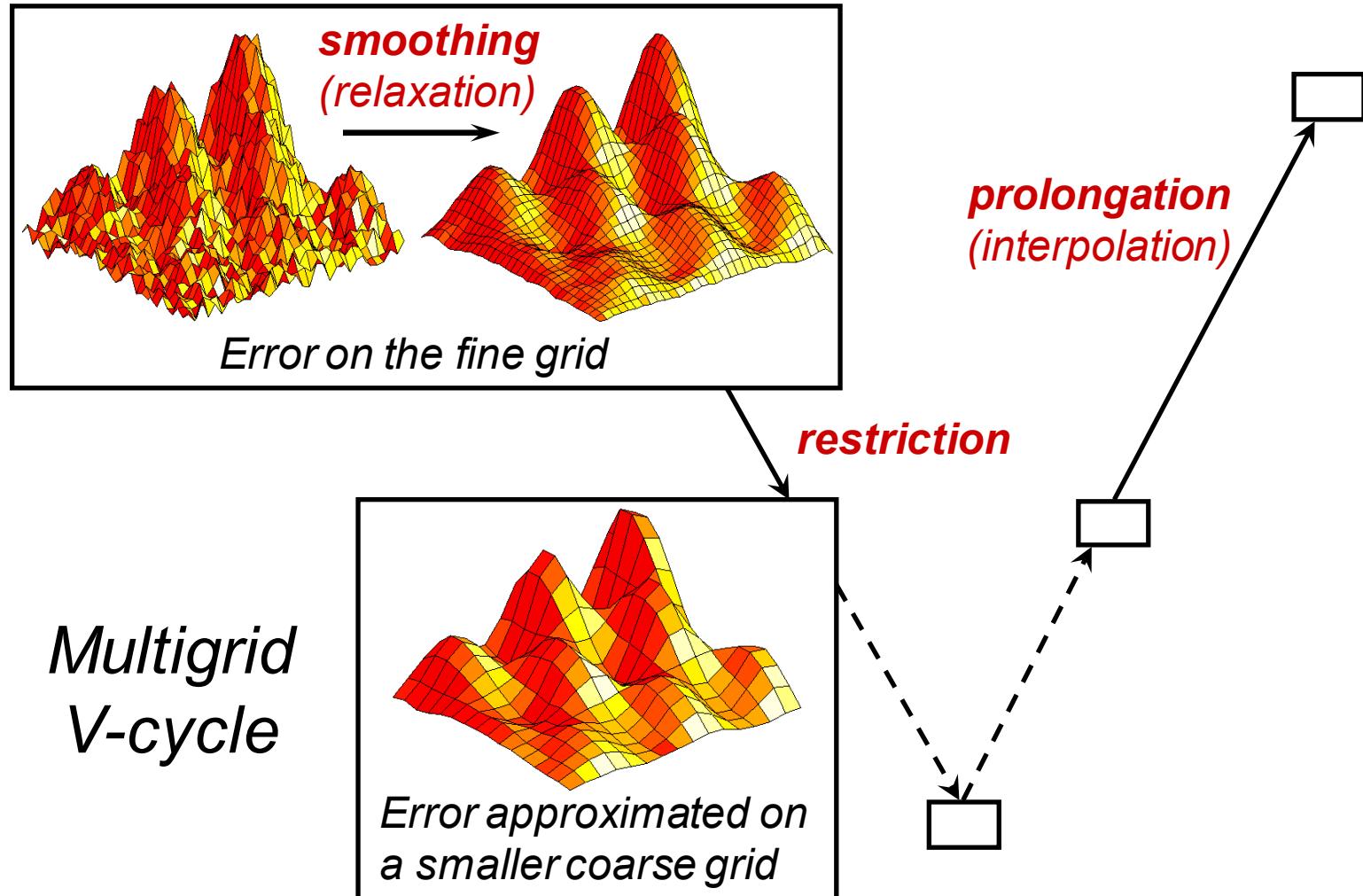


Multigrid solvers have $O(N)$ complexity, and hence have good scaling potential



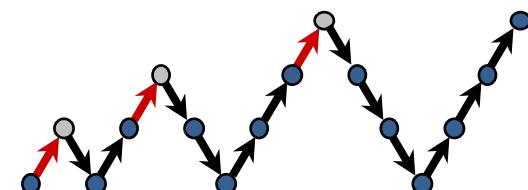
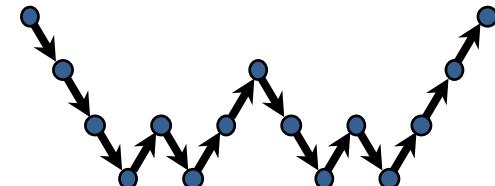
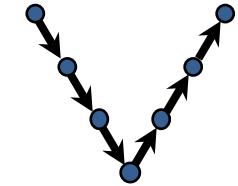
- Weak scaling – want constant solution time as problem size grows in proportion to the number of processors

Multigrid (MG) uses a sequence of coarse grids to accelerate the fine grid solution

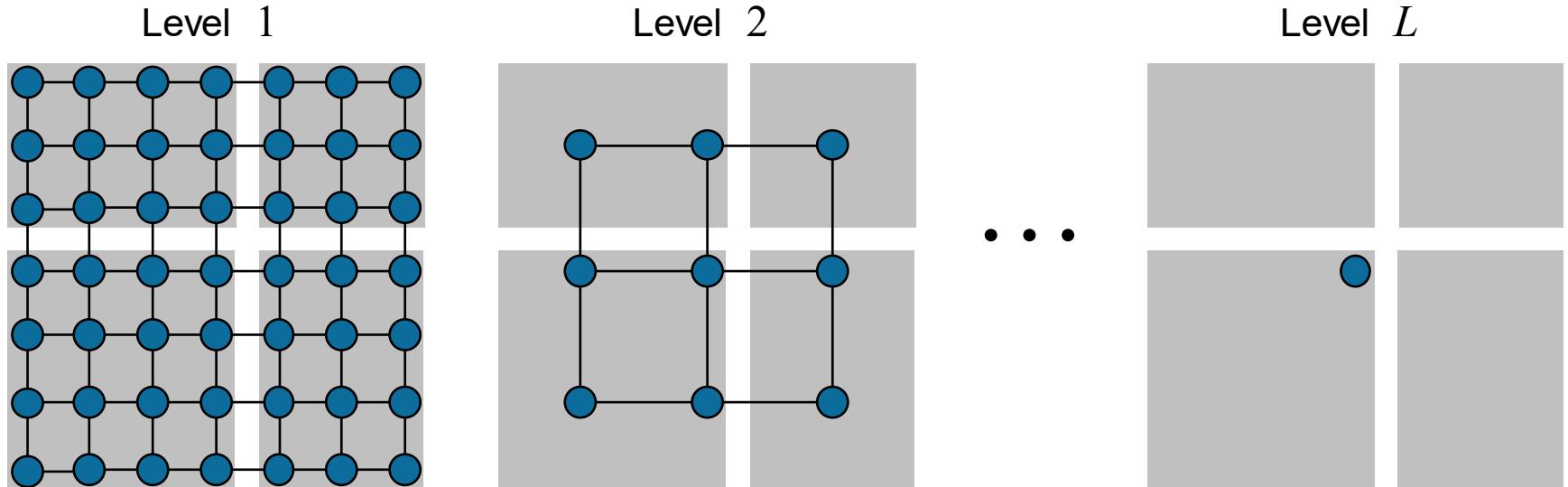


Comments on multigrid cycles

- V-cycle:
 - Most commonly used cycle
 - $O(N)$ work satisfies $\|e\| \leq \varepsilon$ for fixed tolerance ε
- W-cycle:
 - More robust than V-cycles
 - $O(N)$ work satisfies $\|e\| \leq \varepsilon$
 - Not scalable in parallel (discussed later)
- FMG V-cycle:
 - $O(N)$ work satisfies $\|e\| \leq kh^p$ where h^p is discretization accuracy

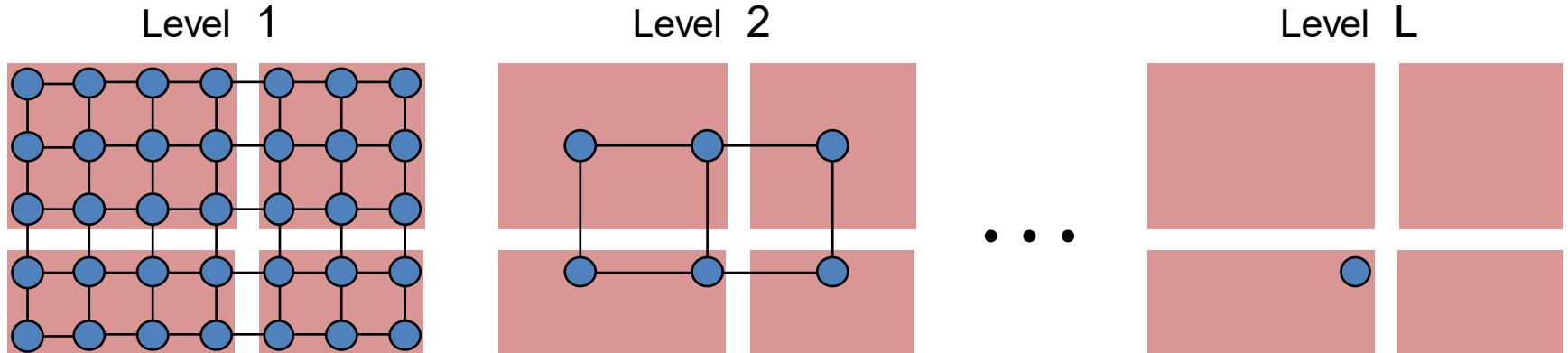


Approach for parallelizing multigrid is straightforward data decomposition



- Basic communication pattern is “nearest neighbor”
 - Relaxation, interpolation, & Galerkin not hard to implement
- Different neighbor processors on coarse grids
- Many idle processors on coarse grids (1.5M+ on BG/Q)
 - Algorithms to take advantage have had limited success

Straightforward MG parallelization yields optimal-order performance for V-cycles



- ~ 1.5 million idle cores on Sequoia!
- Multigrid has a high degree of concurrency
 - Size of the **sequential component** is only $O(\log N)$!
 - This is often the **minimum size achievable**
- Parallel performance model has the expected log term
$$T_V = O(\log N)(\text{comm latency}) + O(\Gamma_p)(\text{comm rate}) + O(\Omega_p)(\text{flop rate})$$

Straightforward parallelization approach is optimal for V-cycles on structured grids (5-pt Laplacian example)

- Standard communication / computation models

$$T_{comm} = \alpha + m\beta \quad (\text{communicate } m \text{ doubles})$$

$$T_{comp} = m\gamma \quad (\text{compute } m \text{ flops})$$

- Time to do relaxation

$$T \approx 4\alpha + 4n\beta + 5n^2\gamma$$

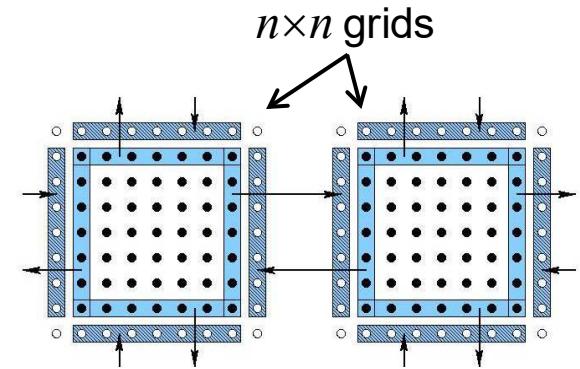
- Time to do relaxation in a V(1,0) multigrid cycle

$$\begin{aligned} T_V &\approx (1 + 1 + \dots)4\alpha + (1 + 1/2 + \dots)4n\beta + (1 + 1/4 + \dots)5n^2\gamma \\ &\approx (\log N)4\alpha + (2)4n\beta + (4/3)5n^2\gamma \end{aligned}$$

- For achieving optimality in general, the *log* term is unavoidable!

- More precise:

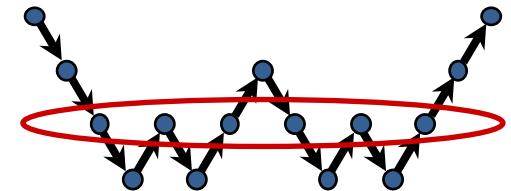
$$T_{V,better} \approx T_V + (\log P)(4\beta + 5\gamma)$$



Additional comments on parallel multigrid

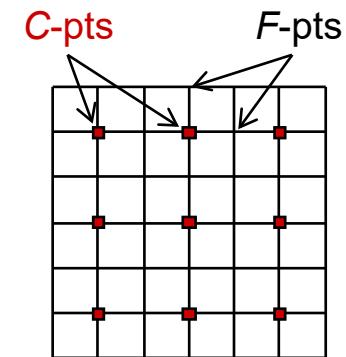
- W-cycles scale poorly:

$$T_W \approx (2^{\log N})4\alpha + (\log N)4n\beta + (2)5n^2\gamma$$



- Lexicographical Gauss-Seidel is too sequential

- Use red/black or multi-color GS
- Use weighted Jacobi, hybrid Jacobi/GS, L1
- Use C-F relaxation (Jacobi on C-pts then F-pts)
- Use Polynomial smoothers



- Parallel smoothers are often less effective
 - More details on this in the section on parallel AMG

- Survey on parallel multigrid, paper on parallel smoothers:
 - ["A Survey of Parallelization Techniques for Multigrid Solvers,"](#) Chow, Falgout, Hu, Tuminaro, and Yang, *Parallel Processing For Scientific Computing*, Heroux, Raghavan, and Simon, editors, SIAM, series on Software, Environments, and Tools (2006)
 - ["Multigrid Smoothers for Ultra-Parallel Computing,"](#) Baker, Falgout, Kolev, and Yang, *SIAM J. Sci. Comput.*, 33 (2011), pp. 2864-2887.

Parallel Algebraic Multigrid (AMG)



AMG Building Blocks

Setup Phase:

- Select coarse “grids”
- Define interpolation: $P^{(m)}$, $m = 1, 2, \dots$
- Define restriction: $R^{(m)}$, $m = 1, 2, \dots$, often $R^{(m)} = (P^{(m)})^T$
- Define coarse-grid operators: $A^{(m+1)} = R^{(m)} A^{(m)} P^{(m)}$

Galerkin product

Solve Phase:

Relax $A^{(m)}u^m = f^m$

Compute $r^m = f^m - A^{(m)}u^m$

Restrict $r^{m+1} = R^{(m)}r^m$

$A^{(m+1)}e^{m+1} = r^{m+1}$

Correct

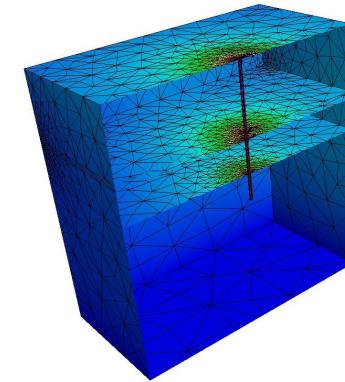
$u^m \leftarrow u^m + e^m$

Interpolate $e^m = P^{(m)}e^{m+1}$

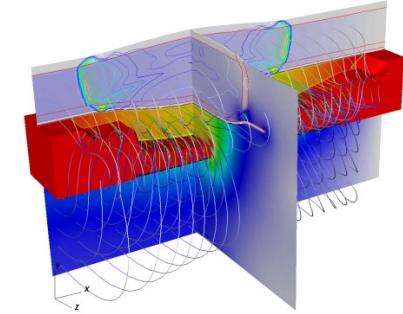
Solve

Parallel computing imposes restrictions on multigrid algorithm development

- Avoid sequential techniques
 - Classical AMG coarsening
 - Gauss-Seidel smoother
 - Cycles with large sequential component
 - F-cycle: $O(\log^2 N)$
 - W-cycle: $O(2^{\log N}) = O(N)$
- Control communication
 - Galerkin coarse-grid operators (P^TAP) can lead to high communication costs in AMG
- Need both CS and Math advances!
 - New methods have new convergence and robustness characteristics
 - Successful addressing issues so far



10x speedup for subsurface problems with new coarsening and interpolation approach



Magnetic flux compression generator simulation enabled by MG smoother research

Smoothers in Multigrid

- Multigrid linear solvers are optimal ($O(N)$ operations), and hence have good scaling potential
- One critical component of MG is the **smoother**
- Some of the best smoothers do not parallelize well (e.g., lex Gauss-Seidel)
- Others used today still depend on parallelism and may break down on **millions of processors**

Original Smoother: Gauss-Seidel – sequential!

Problem:

- solving: $\mathbf{Ax}=\mathbf{b}$, \mathbf{A} symmetric positive definite
- smoothing: $\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{M}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}_n)$

$$\mathbf{A} = \begin{pmatrix} & & & \\ & \text{red} & & \\ & & \text{red} & \\ & & & \ddots \\ & & & \\ & & & \text{red} & \end{pmatrix}$$

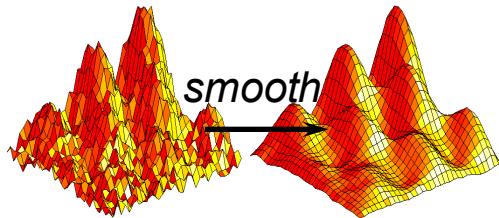
Weighted Jacobi smoother: $\mathbf{M} = \omega\mathbf{D}$

Hybrid Gauß-Seidel smoothers:

- Depends on number of processes
- Might not converge
- Can be improved with an outer weight

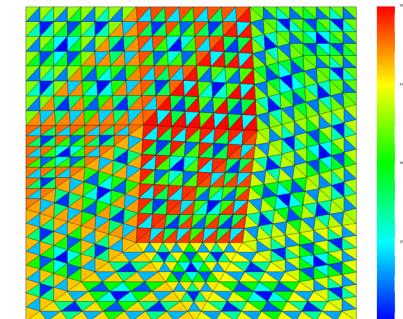
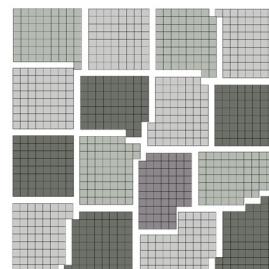
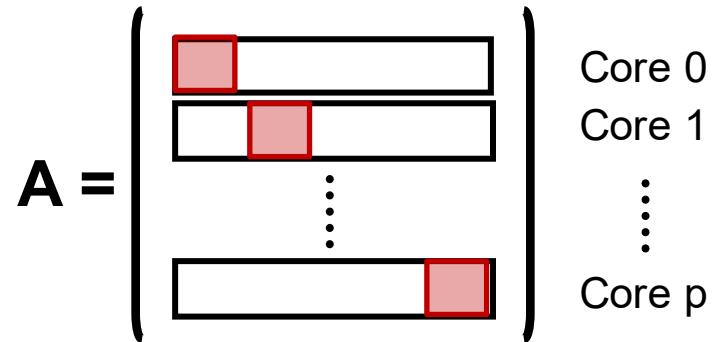
$$\mathbf{M}_H = \begin{pmatrix} \text{red} & & & \\ & \text{red} & & \\ & & \ddots & \\ & & & \text{red} \end{pmatrix}$$

Smoothers significantly affect AMG convergence and runtime.



- **Issues:**
Some of the best smoothers, such as Gauss-Seidel (GS), are highly sequential; others, such as hybrid GS (default smoother in hypre) depend on parallelism
- Convergence can be degraded by:
 - increasing number of blocks
 - decreasing block sizes
 - use of threads (can lead to poor partitioning)

Multi-core/GPU architectures require fine-grained parallelism, less memory, and a different programming model



We pursue two strategies:

1. investigate alternate smoothers , 2. ‘fix’ hybrid smoothers

Alternate smoothers: polynomial smoother $\mathbf{I} - \mathbf{M}^{-1}\mathbf{A} = p(\mathbf{A})$, $p(0)=1$

- independent of parallelism
- MatVec kernel has been tuned
- But need extreme eigenvalue estimates

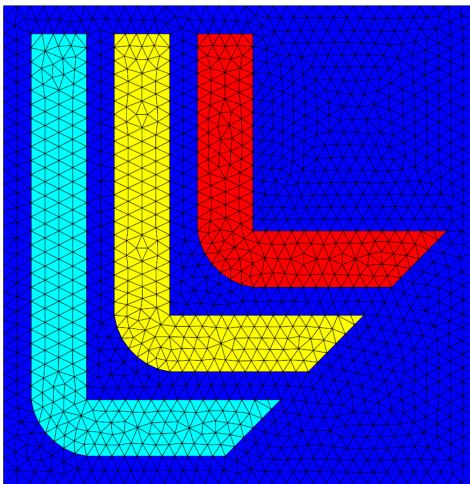
$$\mathbf{A} = \begin{pmatrix} & & & \\ & \text{red} & & \\ & & & \\ & & & \text{red} \\ & & \vdots & \\ & & & & \text{red} & \end{pmatrix}$$

‘Fix’ hybrid smoothers: ℓ_1 – smoother

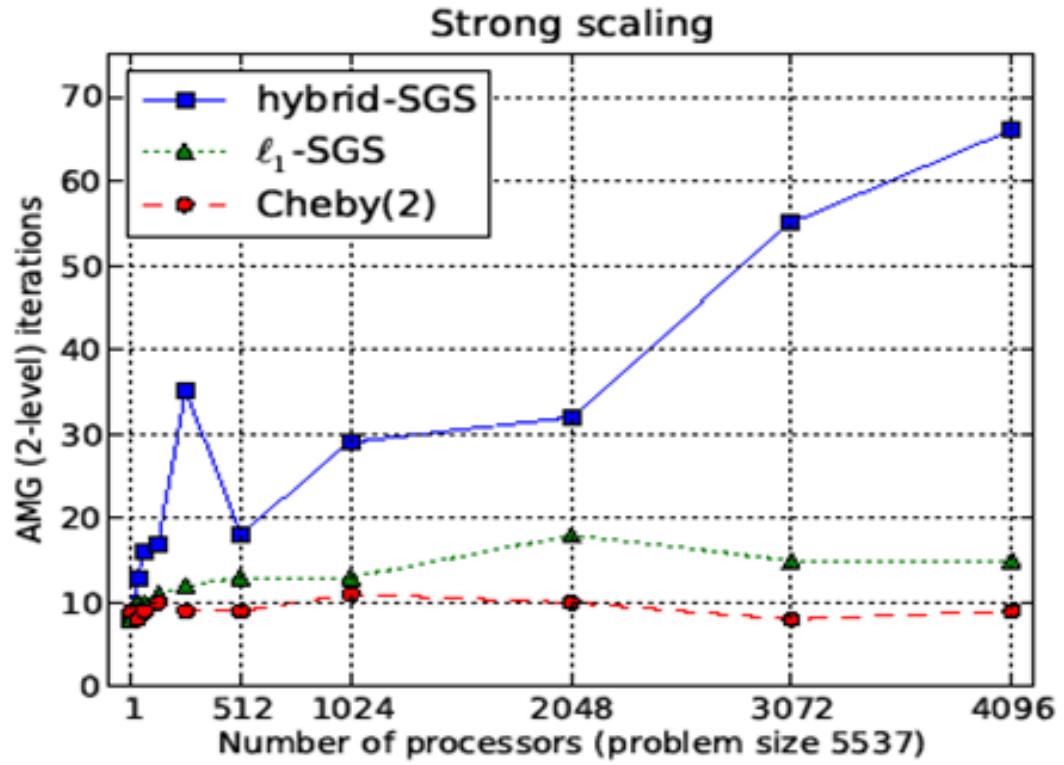
- Add suitable diagonal matrix to hybrid smoother \mathbf{M}_H
$$\mathbf{M}_{\ell_1} = \mathbf{M}_H + \mathbf{D}_{\ell_1} \quad \text{with } \mathbf{D}_{\ell_1} = \sum_{j \neq i} |a_{ij}|^\circ$$
- Always convergent, when GS converges
- Requires no eigenvalue estimates

$$\mathbf{M}_H = \begin{pmatrix} & & & \\ & \text{red} & & \\ & & \text{red} & \\ & & & \text{red} \\ & & \vdots & \\ & & & & \text{red} & \end{pmatrix}$$

Polynomial smoothers and our new ℓ_1 -smoothers are unaffected by decreasing block sizes



Coarse mesh for a diffusion problem with 3 order of magnitude jumps between interior/exterior materials



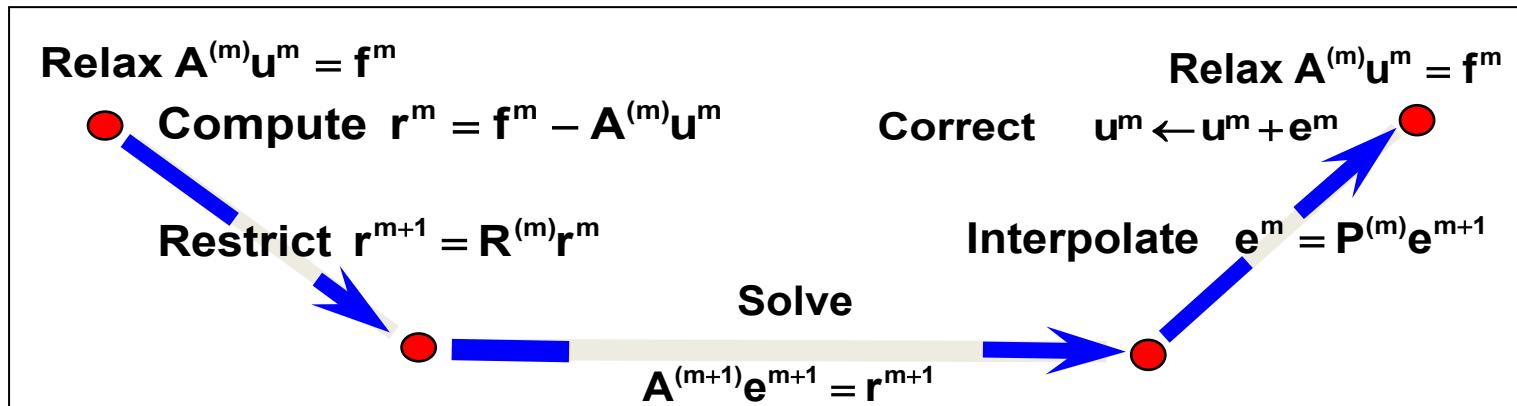
Comments on parallel performance

- Jacobi
 - Parallel (like matvec), processor independent
 - Requires weight which needs to be computed
- L1-Jacobi
 - Parallel, processor independent
 - Always convergent
- C-F Smoothers
 - Parallel if C & F sub-smoothers are
 - Processor independent if C & F sub-smoothers are
- Polynomial Smoothers
 - Parallel, processor independent
 - Requires estimating extreme eigenvalues
- Hybrid Smoothers
 - Parallel and processor dependent
 - Smoothing properties can be processor independent
 - Will break down on coarsest grid levels

Comments on GPU implementation

- Jacobi, polynomial smoothers are all based on matrix-vector multiplications
- MG solve phase can now completely be expressed in terms of matrix-vector multiplications
- Can take advantage of efficient GPU kernels!

Solve Phase:



AMG Setup Phase

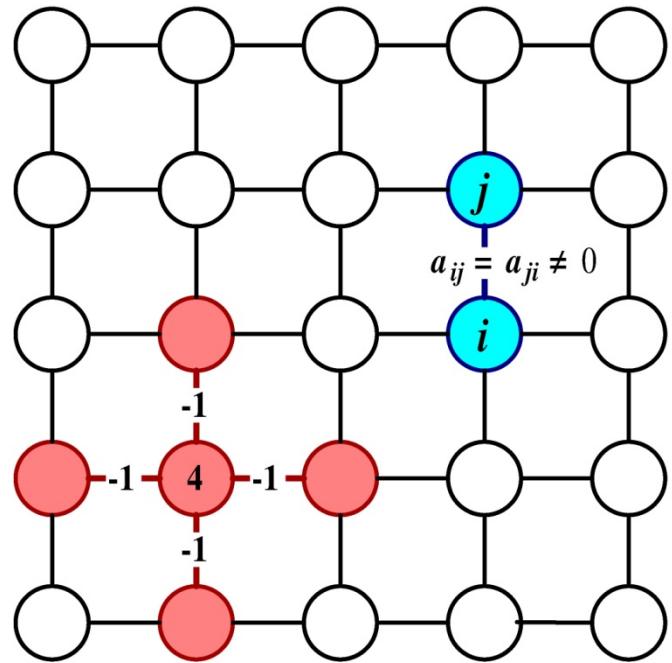
Setup Phase:

- Select coarse “grids”
- Define interpolation: $P^{(m)}$, $m = 1, 2, \dots$
- Define restriction: $R^{(m)}$, $m = 1, 2, \dots$, often $R^{(m)} = (P^{(m)})^T$
- Define coarse-grid operators: $A^{(m+1)} = R^{(m)} A^{(m)} P^{(m)}$



Preliminaries... AMG “grids”

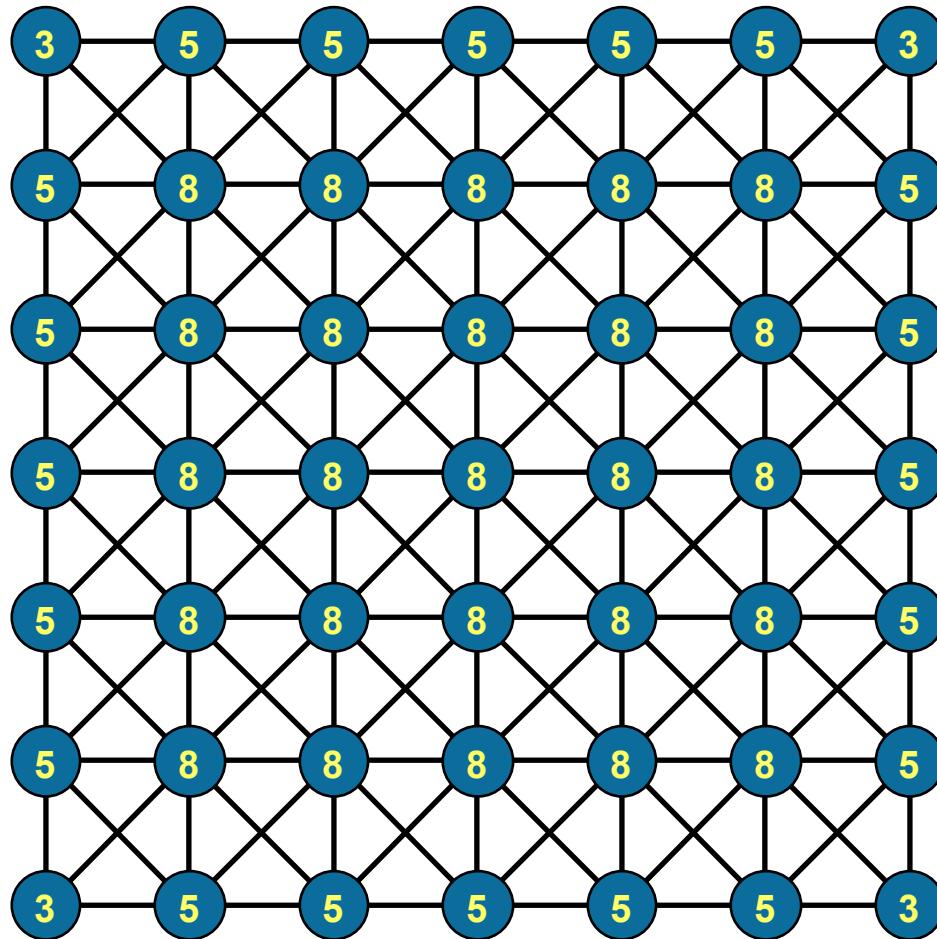
- Matrix adjacency graphs play an important role in AMG:
 - grid = set of graph vertices
 - grid point i = vertex i
- As a **visual aid**, it is highly instructive to relate the matrix equations to an underlying PDE and discretization
- We will often **draw the grid points in their geometric locations**
- Remember that AMG doesn't actually use this geometric information!



Choosing the coarse grid

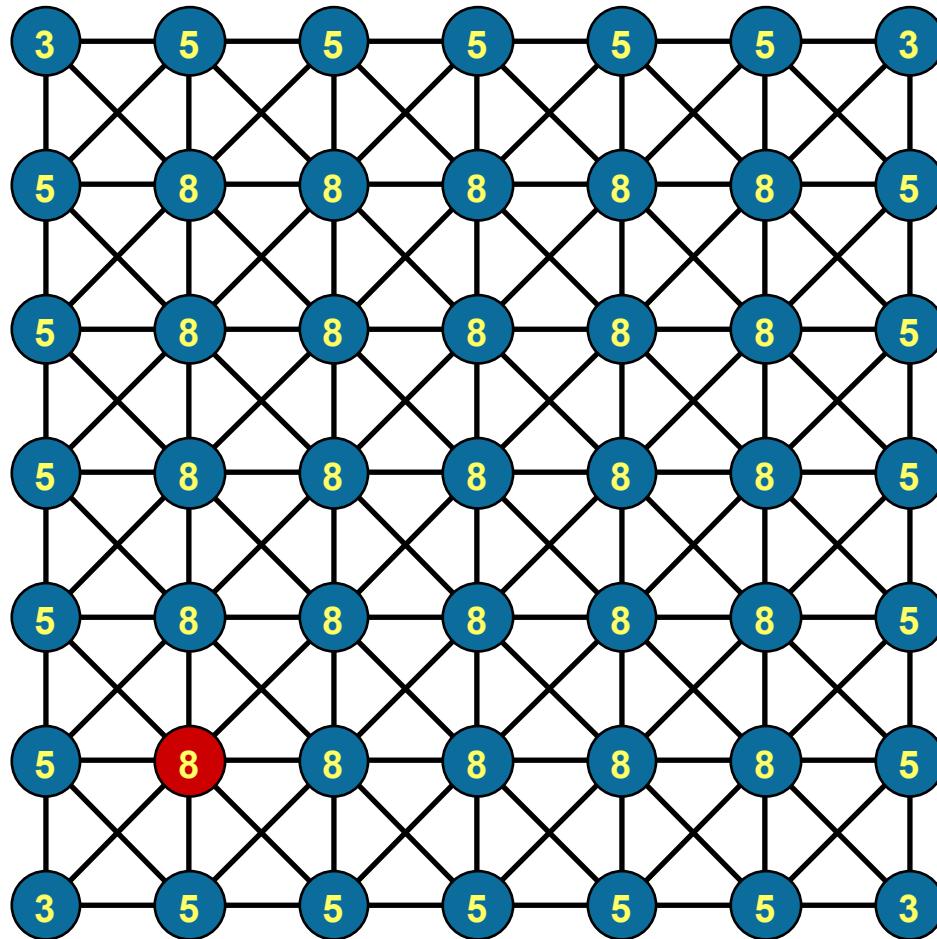
- Smoothed Aggregation AMG uses aggregation of variables to generate a coarse ‘grid’, i.e., set of variables → coarse variable
- Focus here on Classical AMG (C-AMG) – coarse grid is a subset of the fine grid
- The basic coarsening procedure is as follows:
 - Define a **strength matrix** A_s by deleting weak connections in A
 - **First pass:** Choose an independent set of fine-grid points based on the graph of A_s
 - **Second pass:** Choose additional points if needed to satisfy interpolation requirements
- Coarsening partitions the grid into C - and F -points

C-AMG coarsening



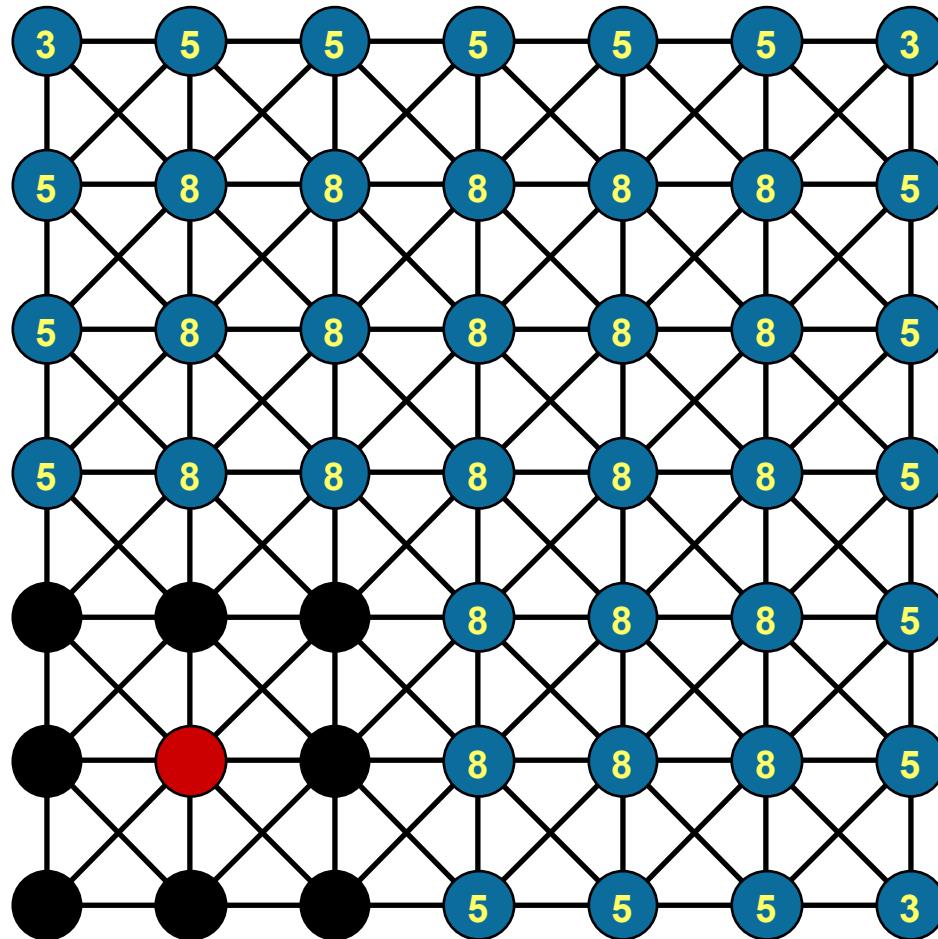
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



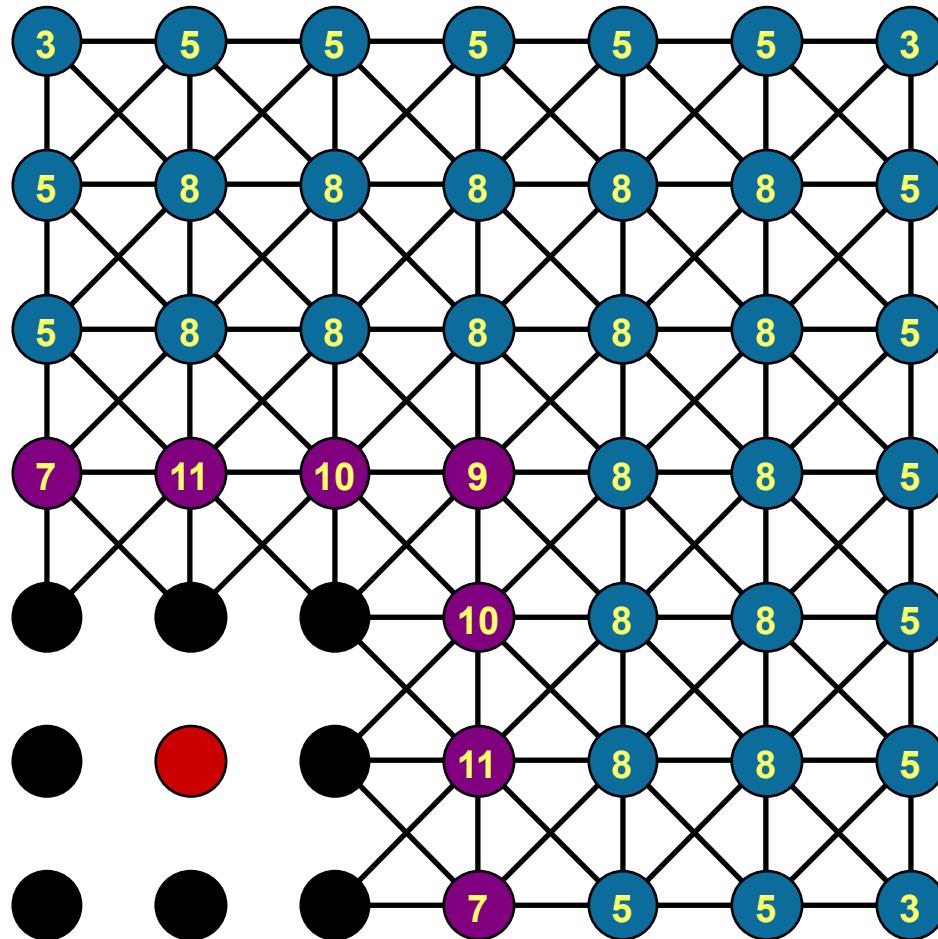
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



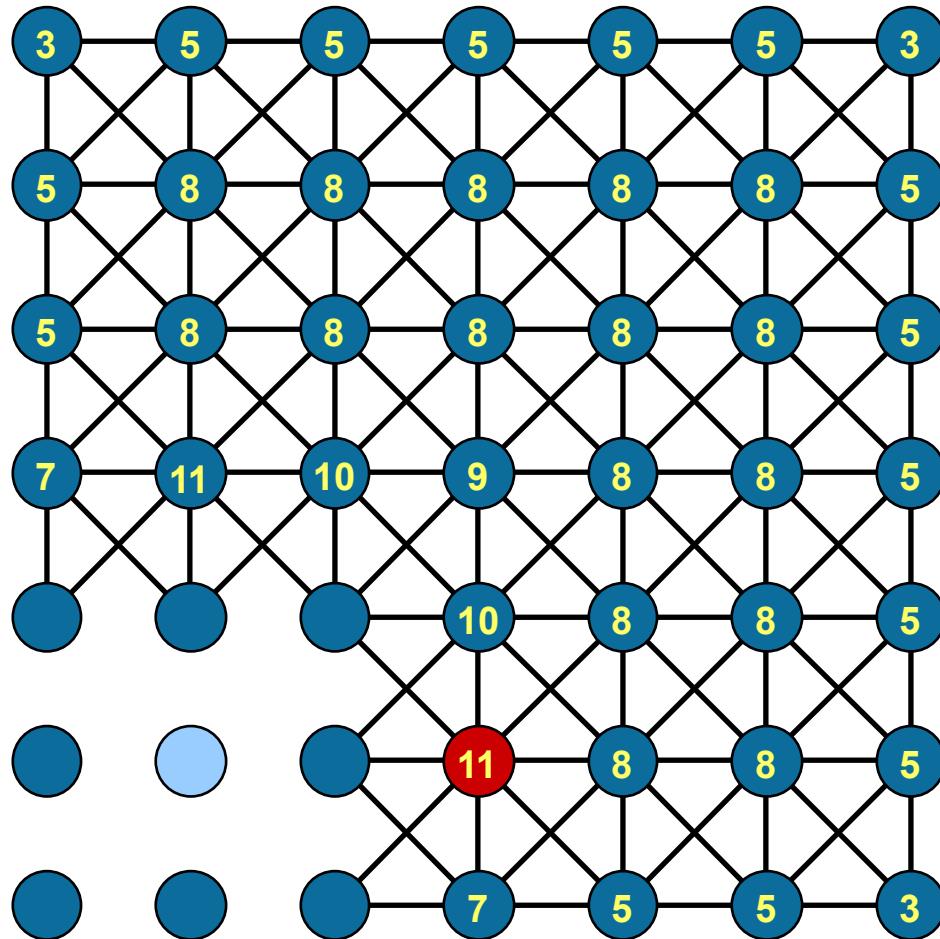
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



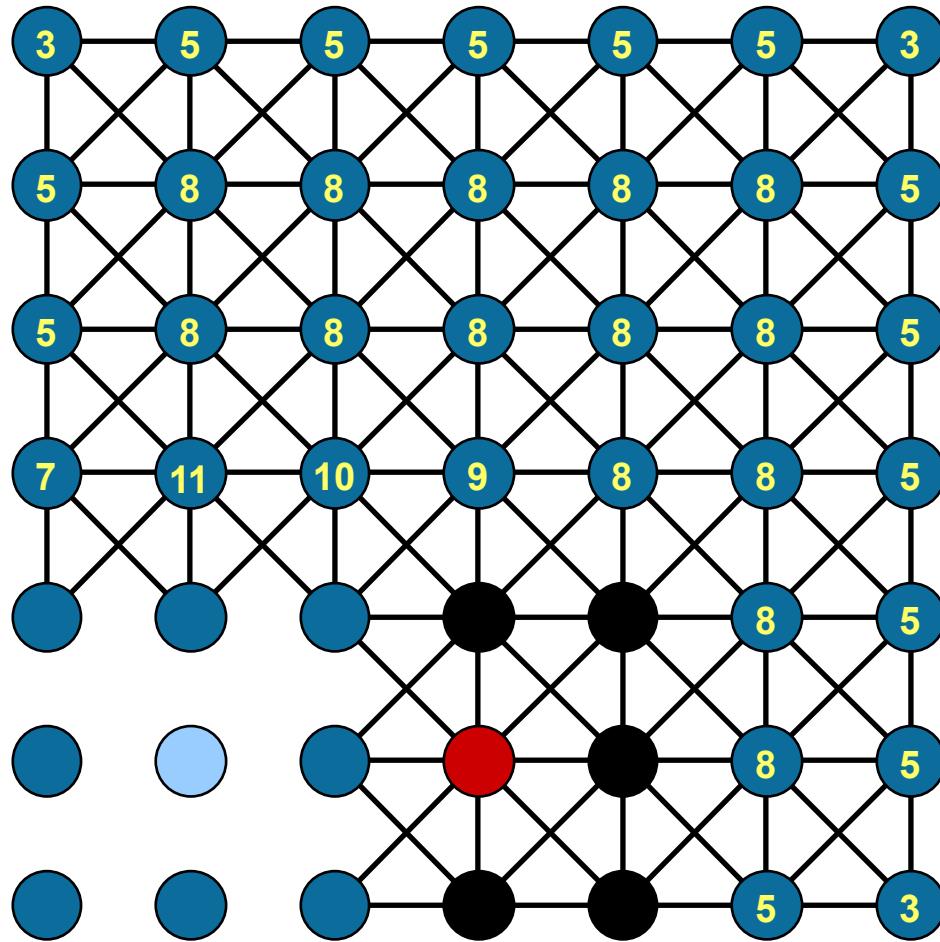
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



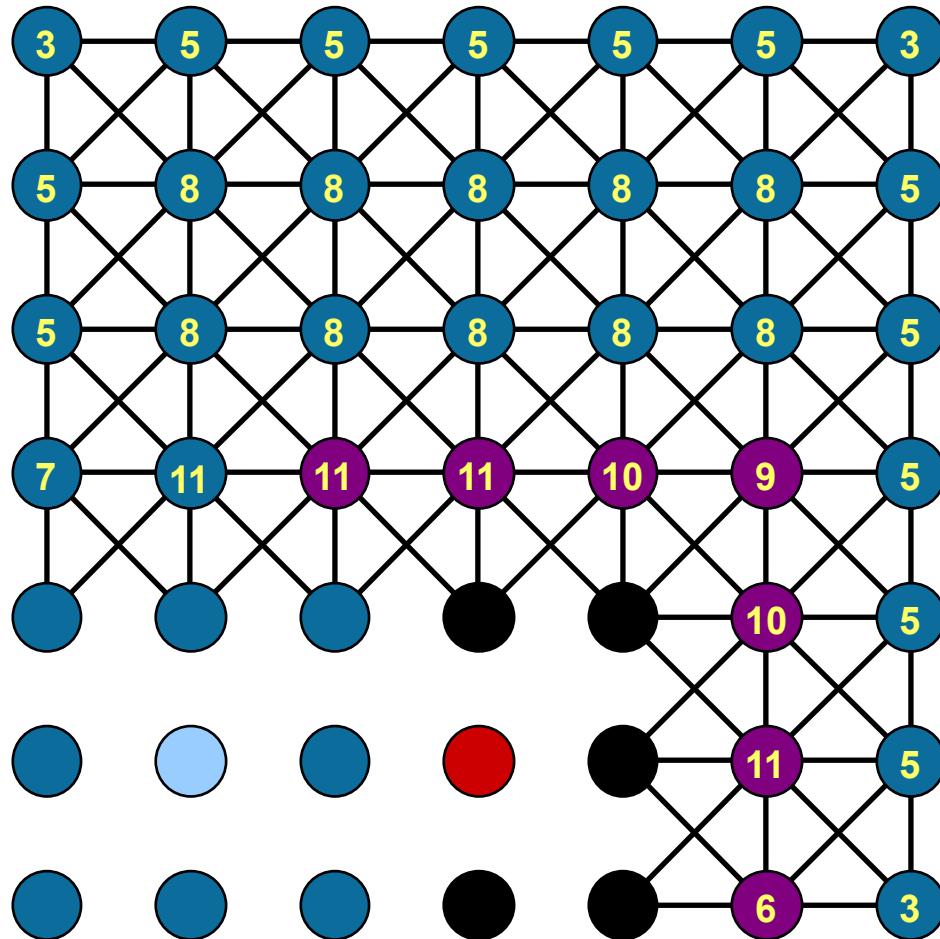
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



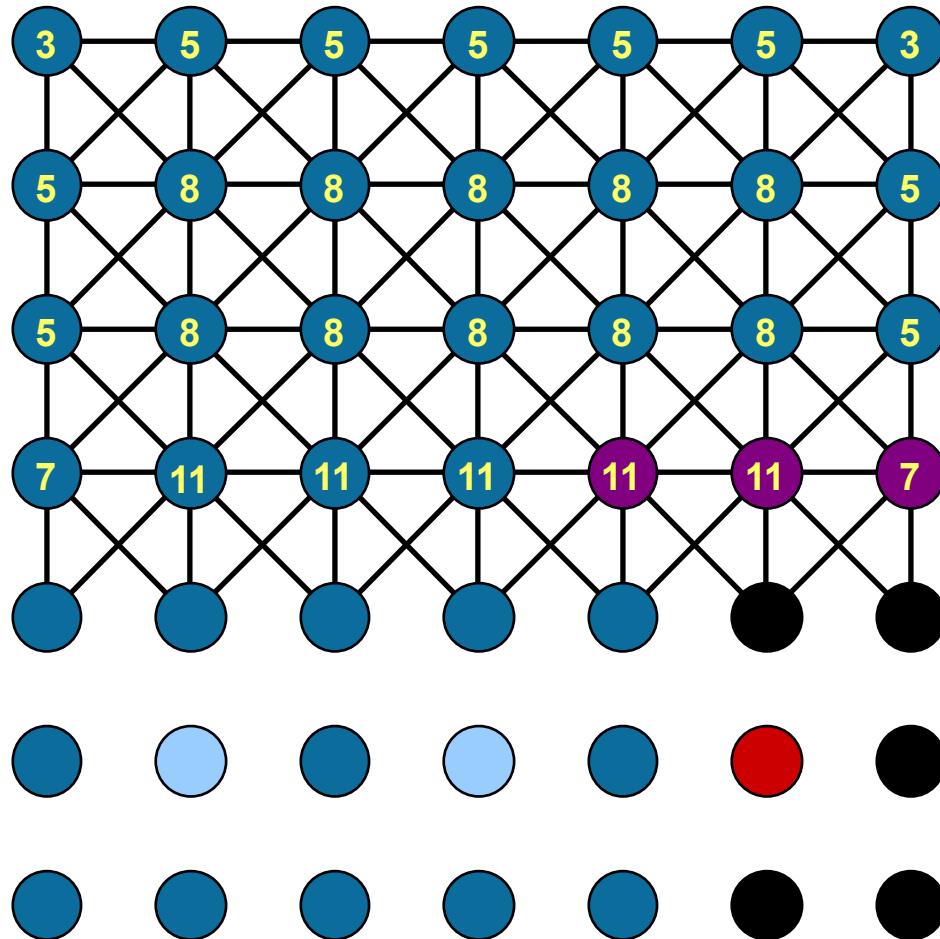
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



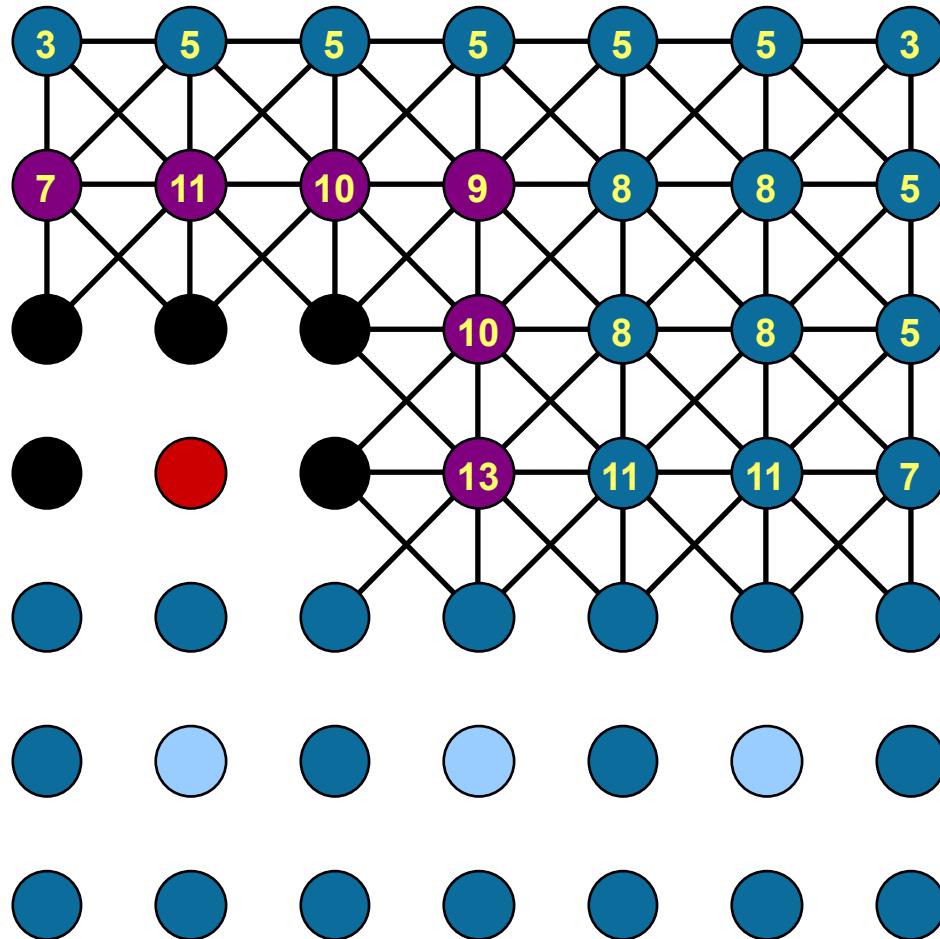
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



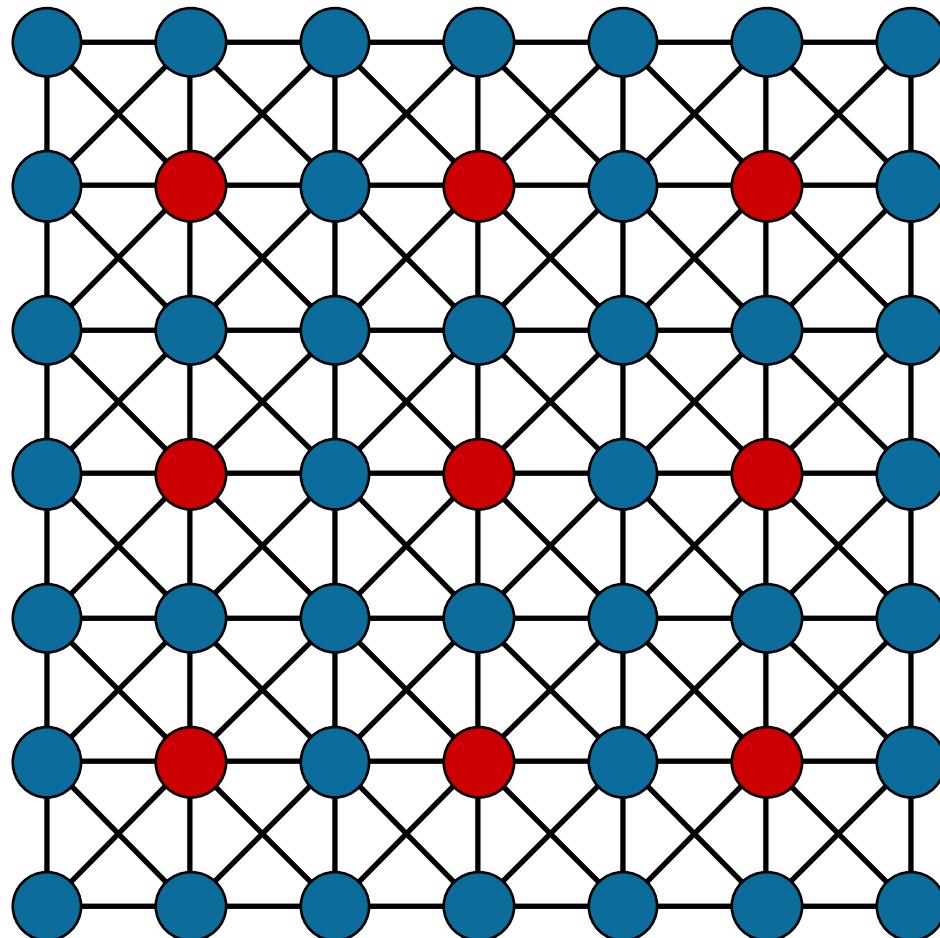
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening



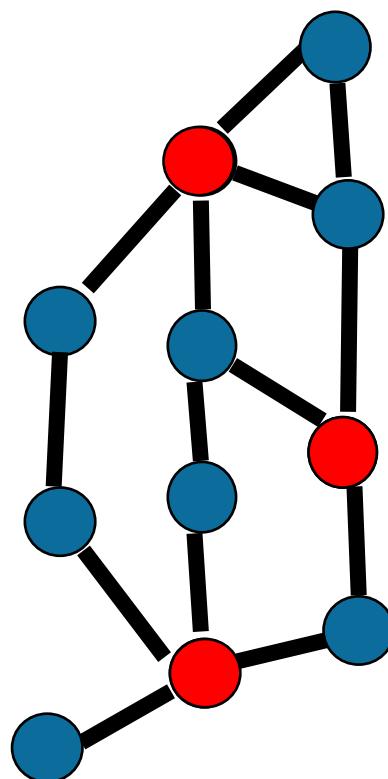
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

C-AMG coarsening is inherently sequential



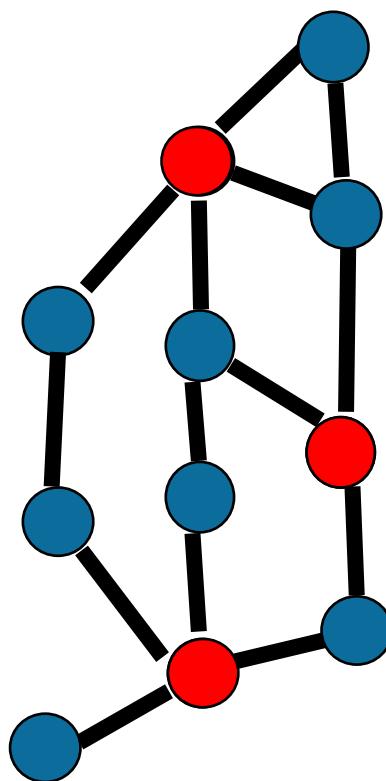
- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

Original AMG coarsening highly sequential!



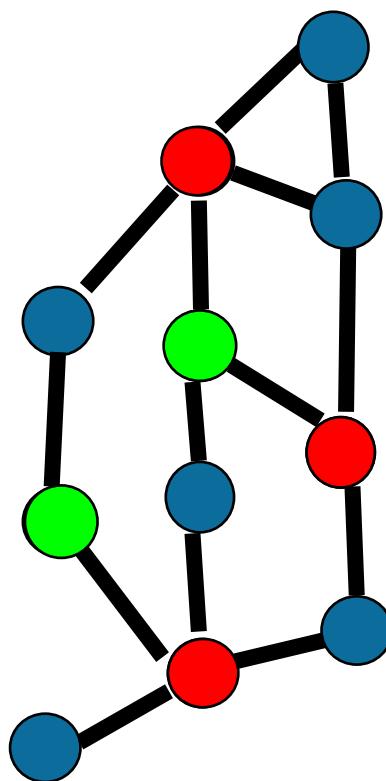
- (C1) Maximal Independent Set:
 - Independent: no two C-points are connected
 - Maximal: if one more C-point is added, the independence is lost

Original AMG coarsening highly sequential!



- (C1) Maximal Independent Set:
Independent: no two **C**-points are connected
Maximal: if one more **C**-point is added, the independence is lost
- (C2) All **F-F** connections require connections to a common **C**-point (for good interpolation)

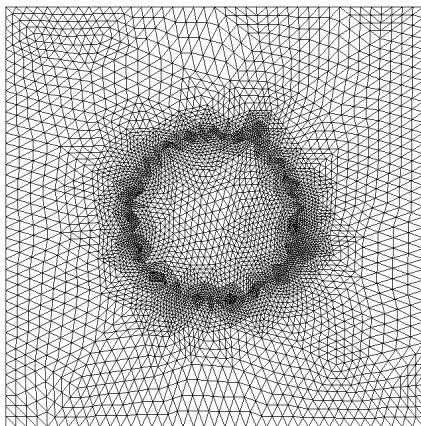
Original AMG coarsening highly sequential!



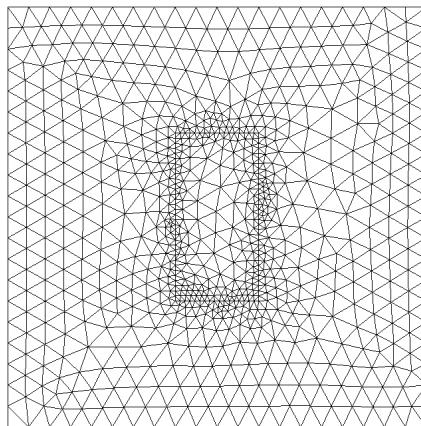
- (C1) Maximal Independent Set:
Independent: no two **C**-points are connected
Maximal: if one more **C**-point is added, the independence is lost
- (C2) All **F-F** connections require connections to a common **C**-point (for good interpolation)
- **F**-points have to be changed into **C**-points, to ensure (C2); (C1) is violated
more C-points, higher complexity

AMG grid hierarchies for several 2D problems

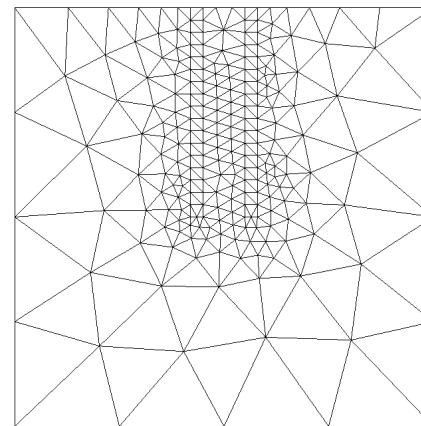
domain1 - 30°



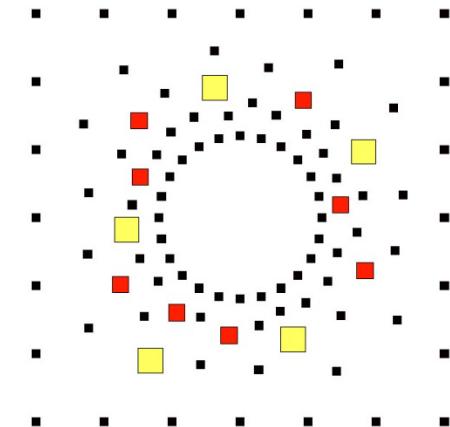
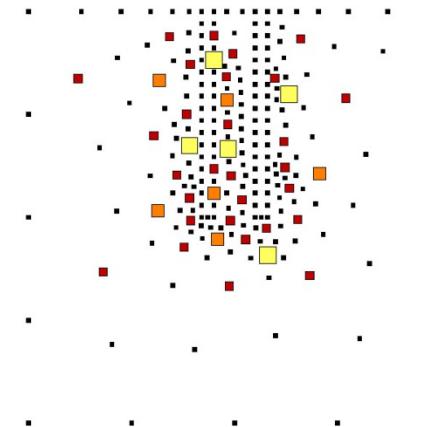
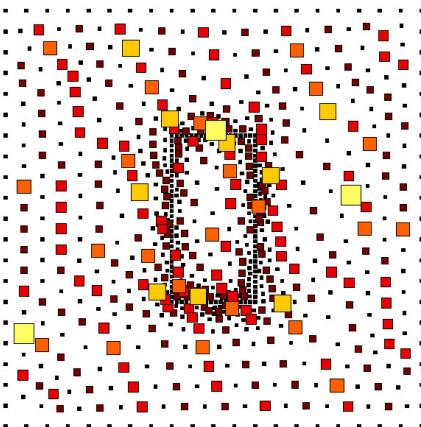
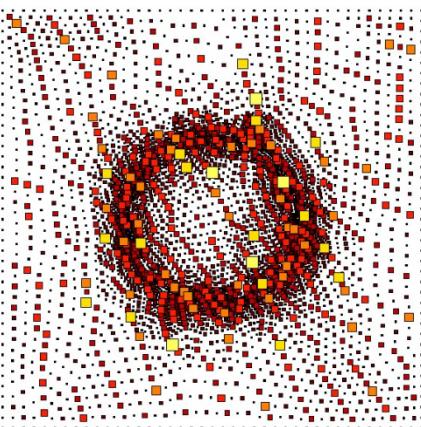
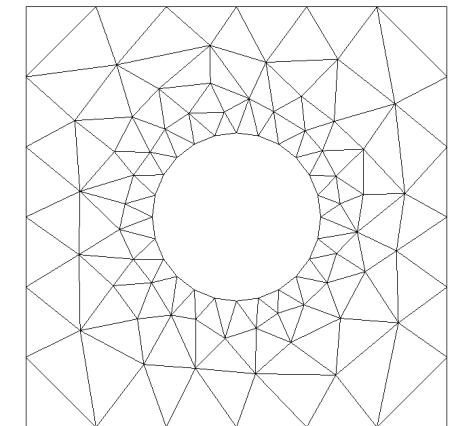
domain2 - 30°



pile



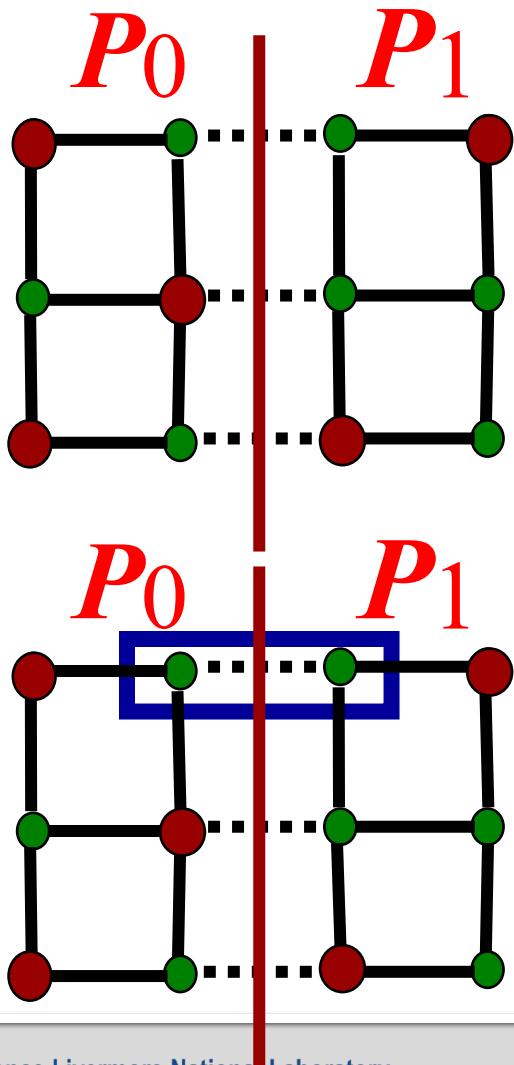
square-hole



Parallel C-AMG Coarsening

- One possible approach to coarsening in parallel: perform the original algorithm on each processor.
- Choice of measures: local or global
- Various treatments possible at processor boundaries.
- Yields processor dependent coarsening and will not produce the same results for different numbers of processors.

Parallel RSO coarsening:

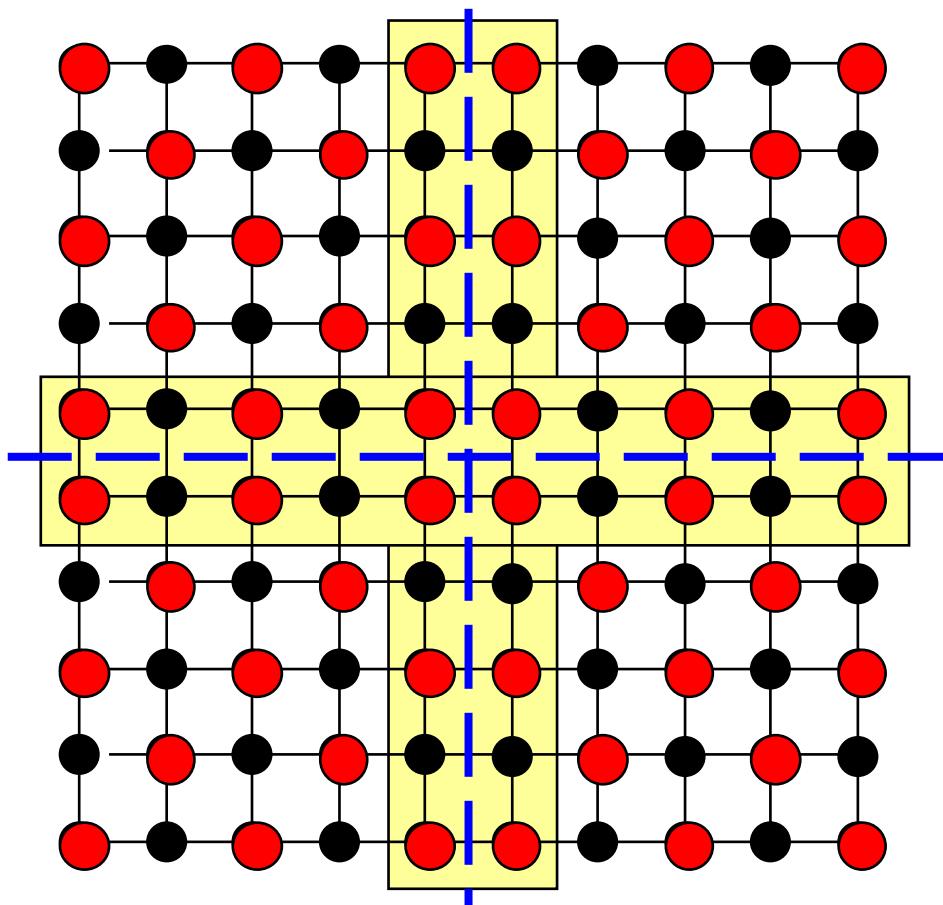


Perform first and second passes on each processor

Method 1: Do nothing.
Accept the coarsening provided by the independent processors.

Problem: Leaves $F \leftrightarrow F$ dependencies without mutual G points

RS0 coarsening, local measures



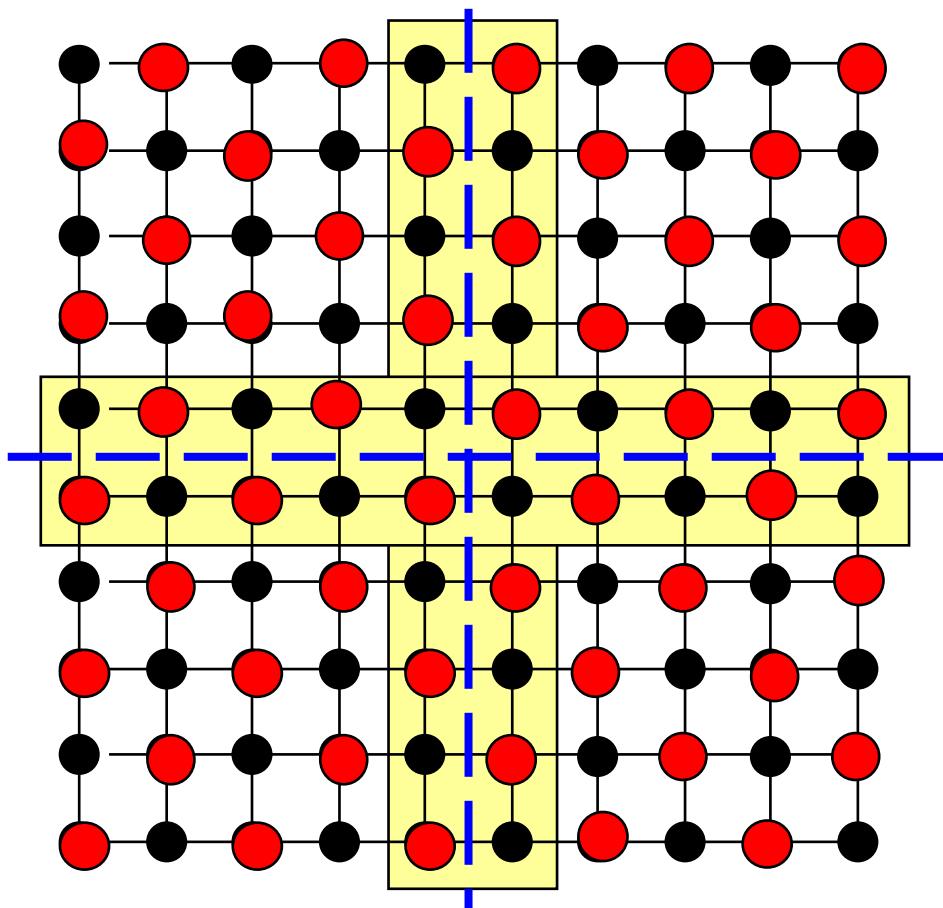
5pt 2d Laplacian

10x10 grid

4 processors

52 C-points

RS0 coarsening, global measures



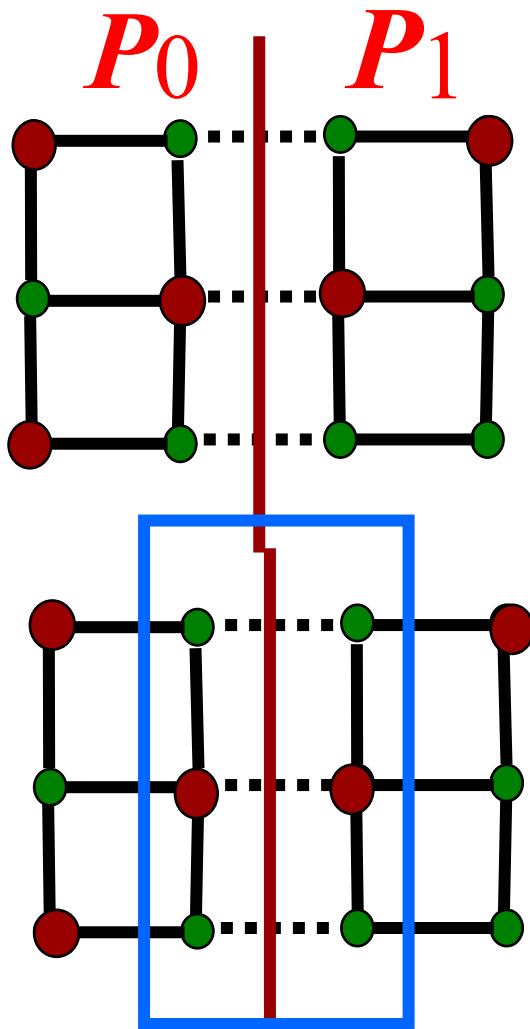
5pt 2d Laplacian

10x10 grid

4 processors

50 C-points

Parallel RS3 coarsening:

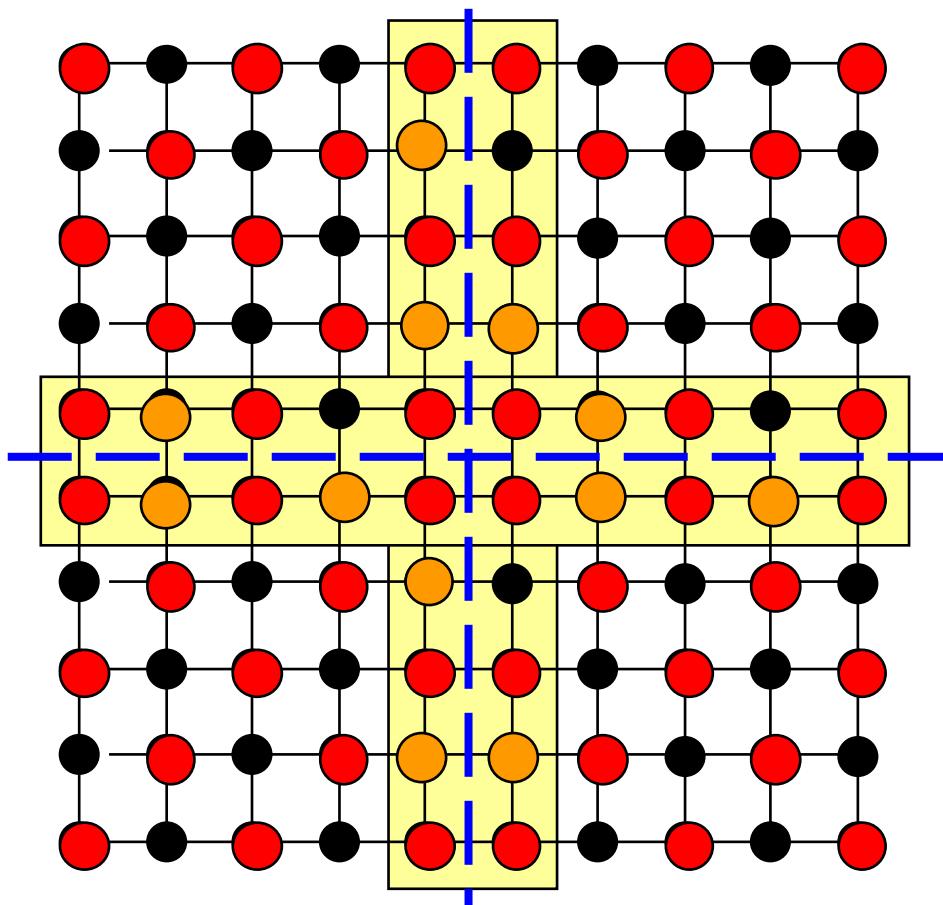


Perform first and second pass on each processor

Perform a third pass, (a second “second pass”), **only on those points adjacent to processor boundaries**

Choices must be made about how to resolve conflicting decisions among processors

RS3 coarsening, local measures



5pt 2d Laplacian

10x10 grid

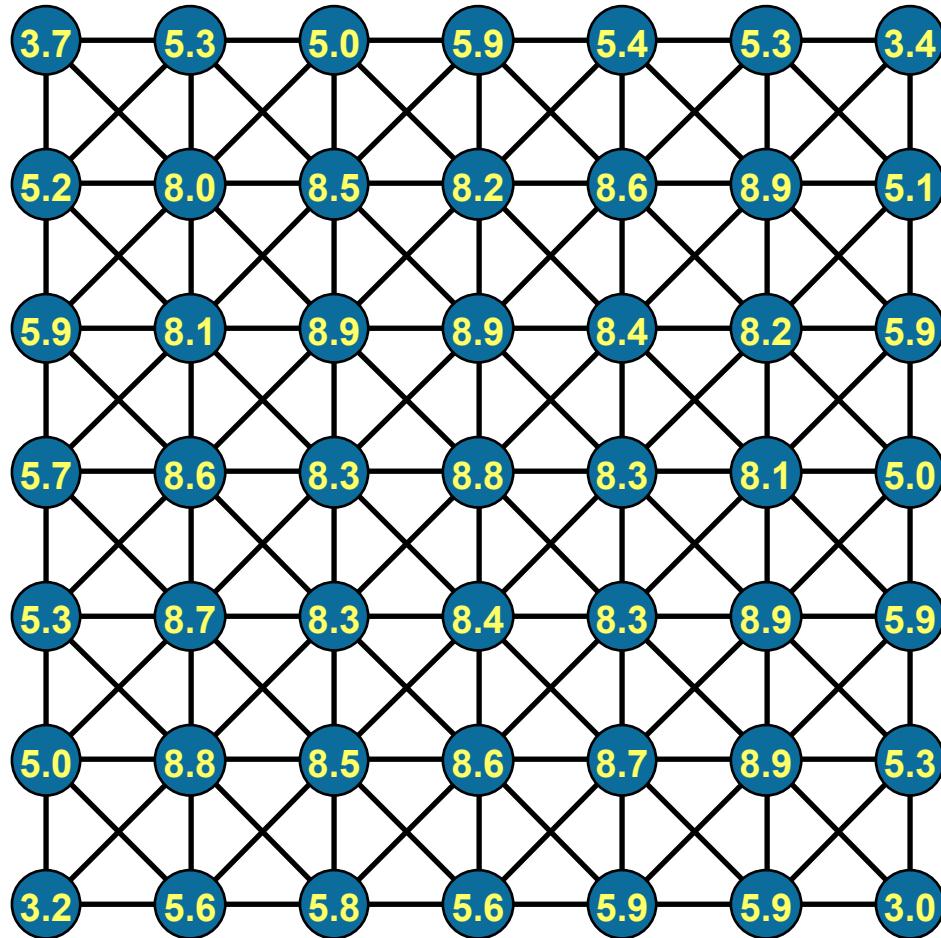
4 processors

64 C-points

CLJP Coarsening Algorithm

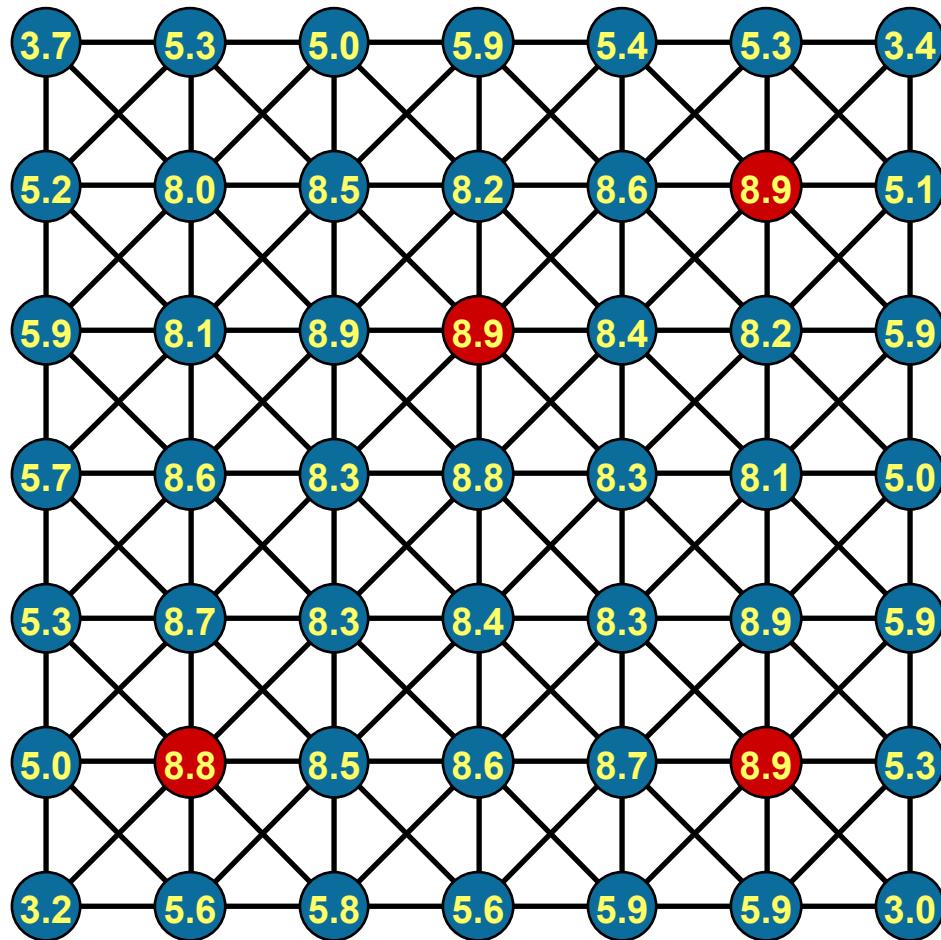
- CLJP (Cleary-Luby-Jones-Plassmann) coarsening
- Based on an algorithm by Jones and Plassmann, which is based on an algorithm by Luby
- Uses one-pass approach with random numbers to get concurrency (illustrated next)

CLJP coarsening is fully parallel



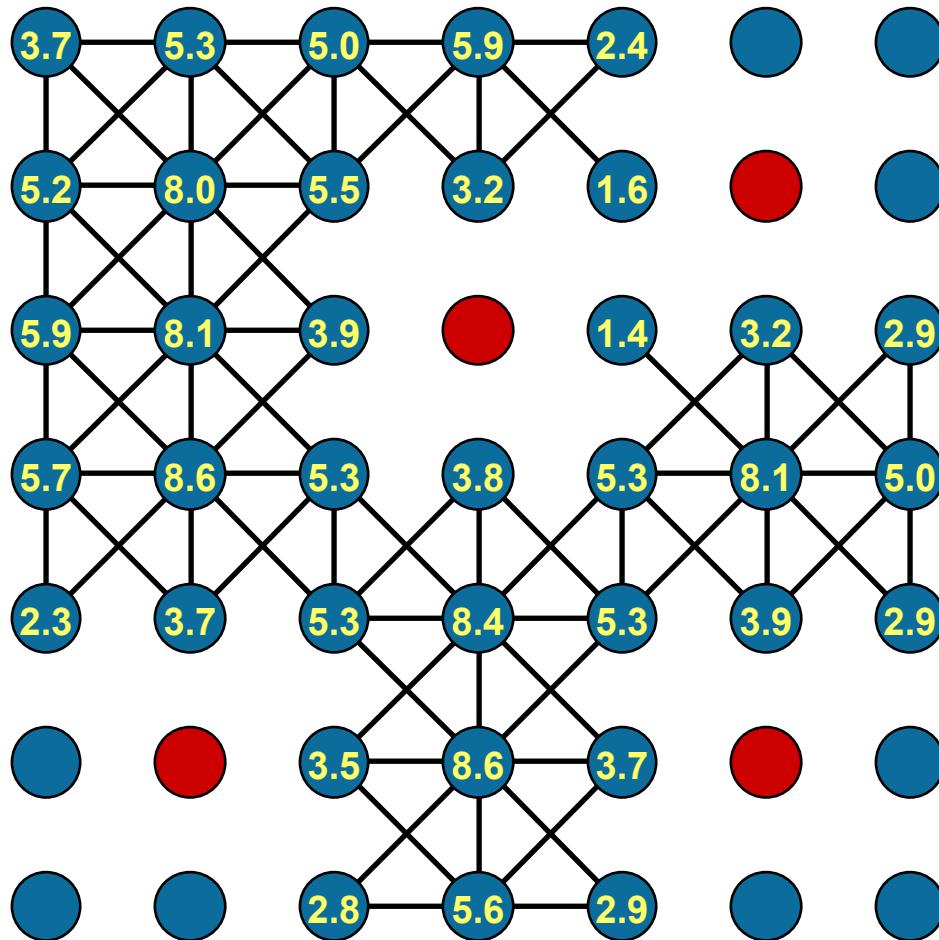
- select C-pts with maximal measure locally
- remove neighbor edges
- update neighbor measures

CLJP coarsening is fully parallel



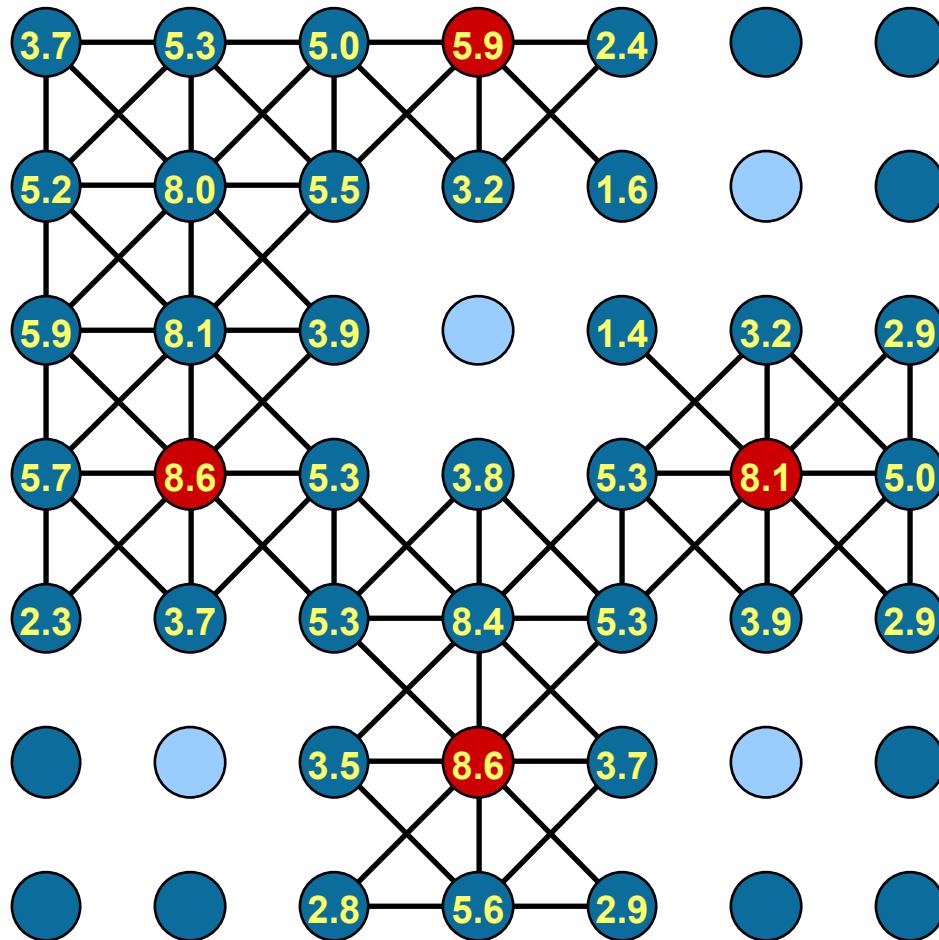
- select C-pts with maximal measure locally
- remove neighbor edges
- update neighbor measures

CLJP coarsening is fully parallel



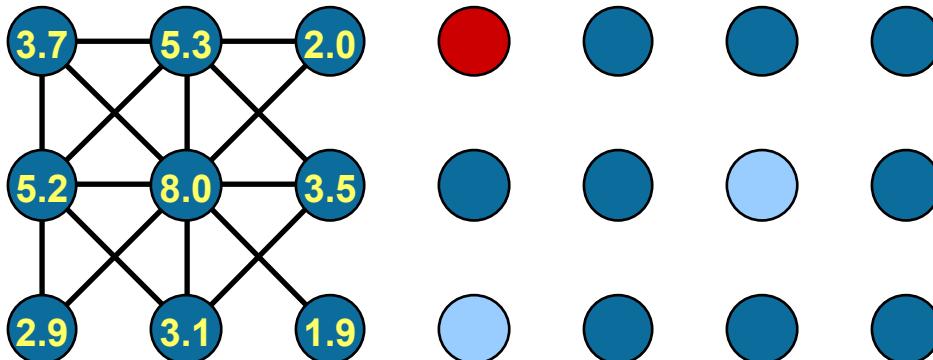
- select C-pts with maximal measure locally
- remove neighbor edges
- update neighbor measures

CLJP coarsening is fully parallel

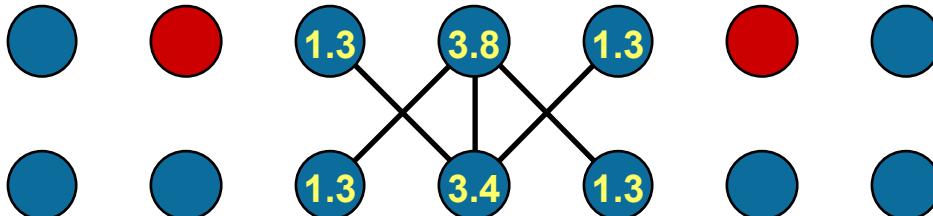


- select C-pts with maximal measure locally
- remove neighbor edges
- update neighbor measures

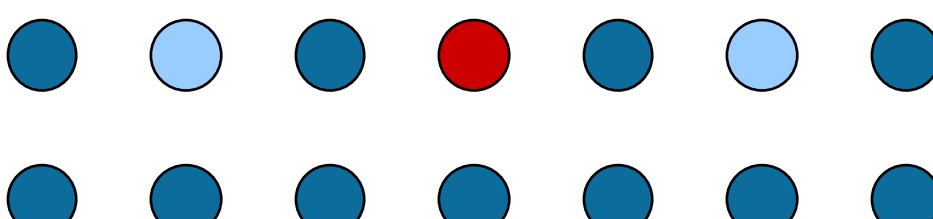
CLJP coarsening is fully parallel



- select C-pts with maximal measure locally

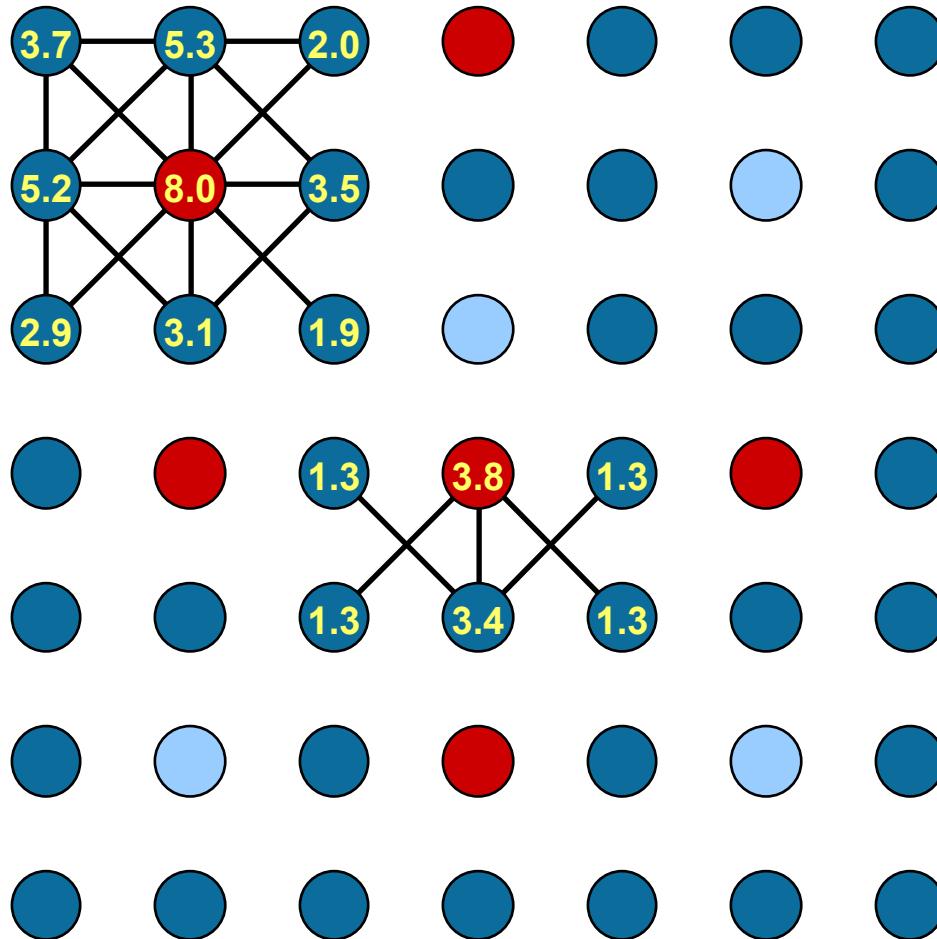


- remove neighbor edges



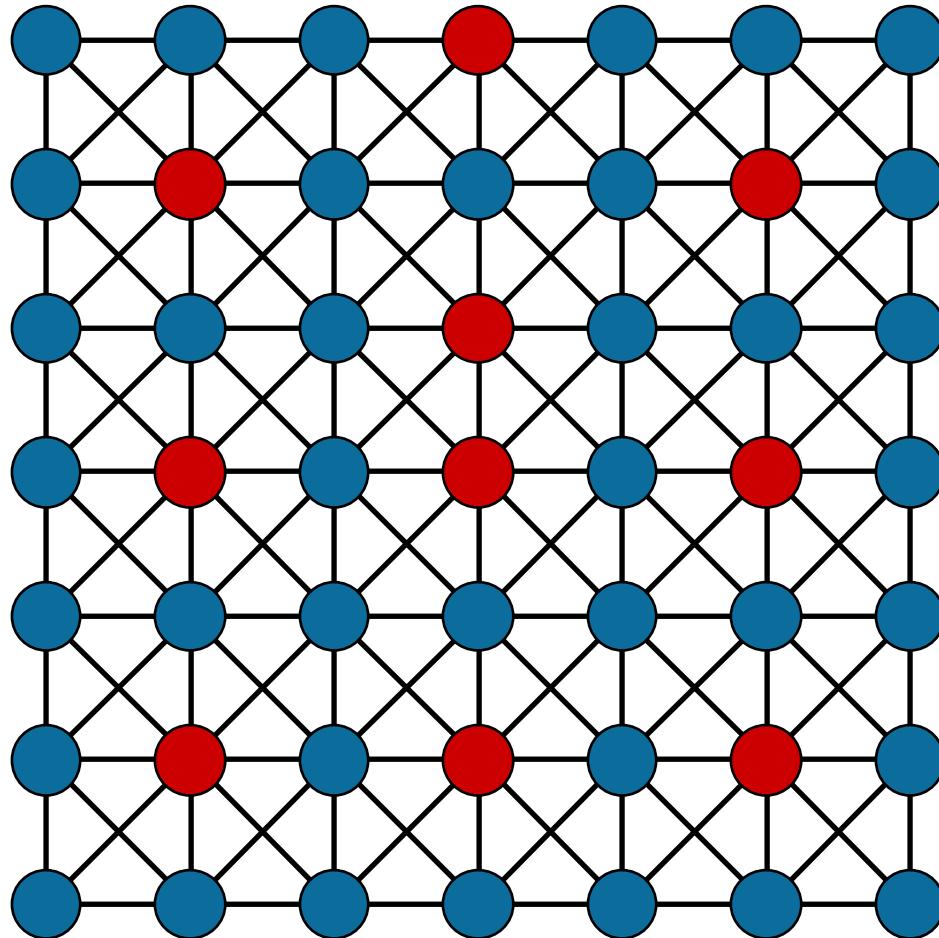
- update neighbor measures

CLJP coarsening is fully parallel



- select C-pts with maximal measure locally
- remove neighbor edges
- update neighbor measures

CLJP coarsening is fully parallel

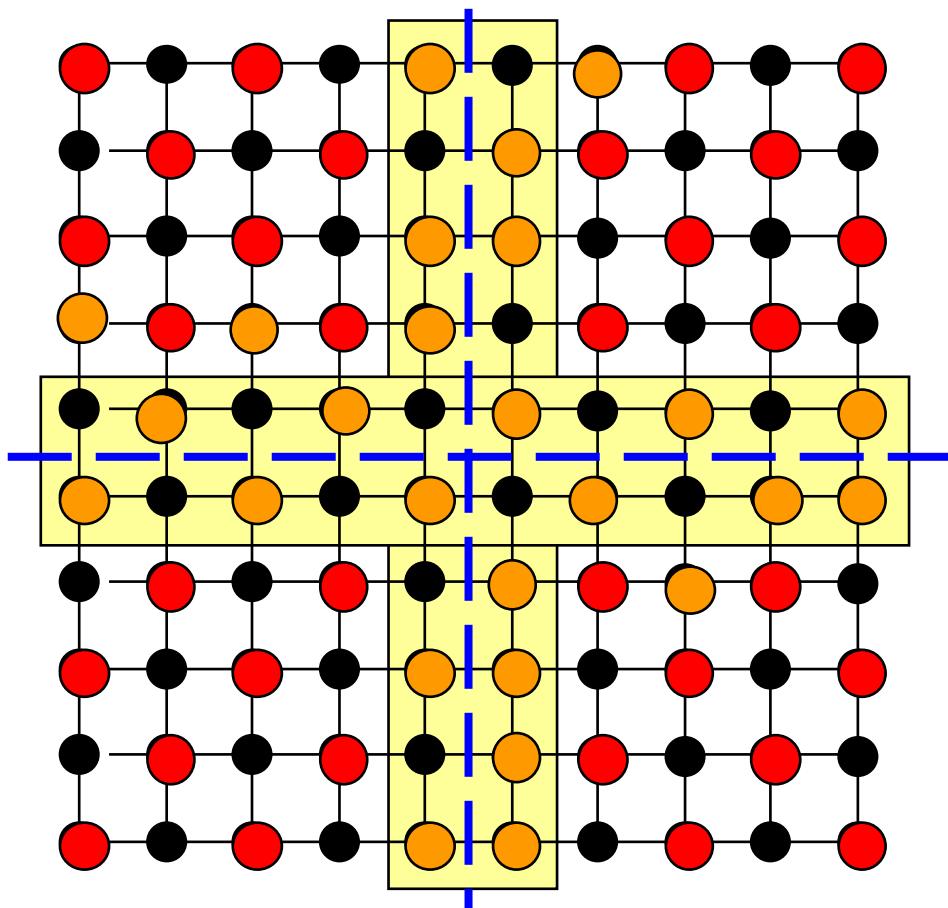


- 10 C-points selected
- Standard AMG selects 9 C-points

Falgout coarsening

- Coming back to our original problem of how to deal with the processor boundaries in parallel original coarsening
- Another coarsening algorithm which combines original and CLJP coarsening was suggested by Falgout
- perform the standard original algorithm on each processor ignoring off processor connections
- apply CLJP algorithm using the so-obtained “interior” coarse points as its first independent set

Falgotout coarsening



5pt 2d Laplacian

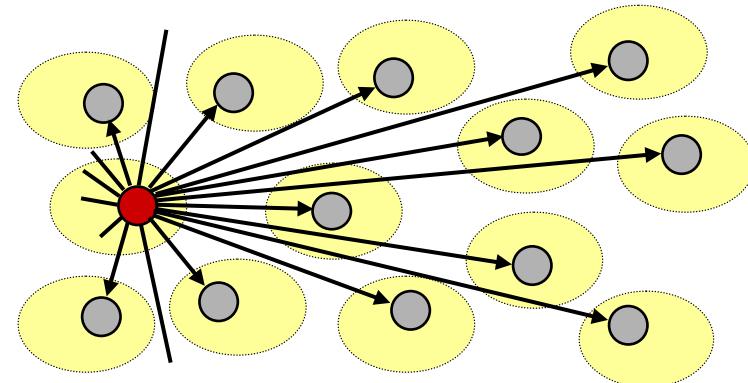
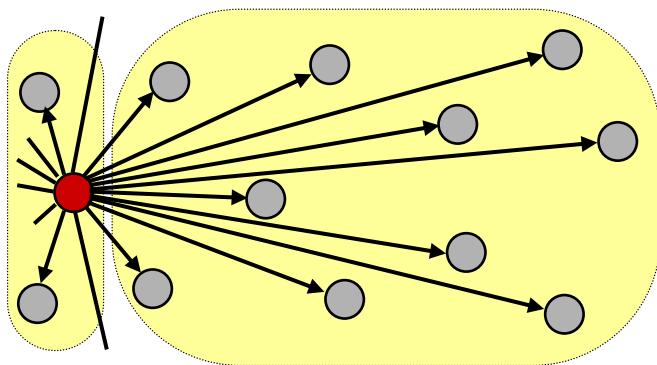
10x10 grid

4 processors

58 C-points

Parallel coarse-grid selection in AMG can produce unwanted side effects

- Second pass in C-AMG coarsenings algorithm might add to many points already for the sequential algorithms, which is exacerbated in parallel
- Non-uniform grids can lead to increased operator complexity and poor convergence
- Operator “stencil growth” reduces parallel efficiency

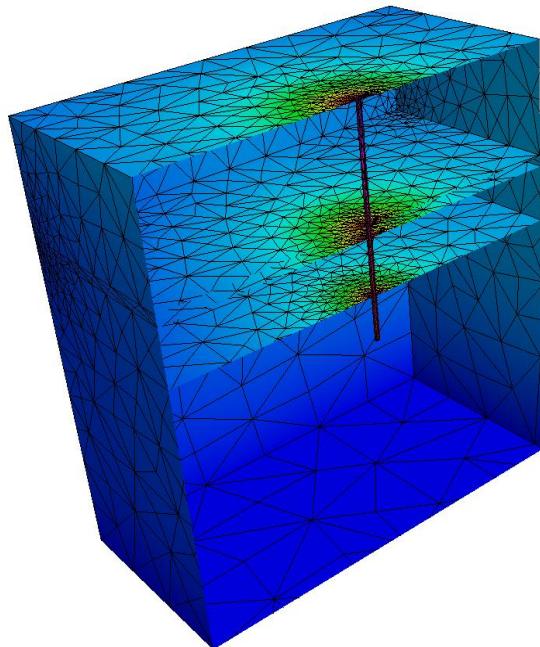


- Solution: coarsen more aggressively!
- Eliminate 2nd pass in original coarsening
- Simplified ‘CLJP’ coarsening → PMIS (parallel maximal independent set)

Unstructured Diffusion Problem with Jumps

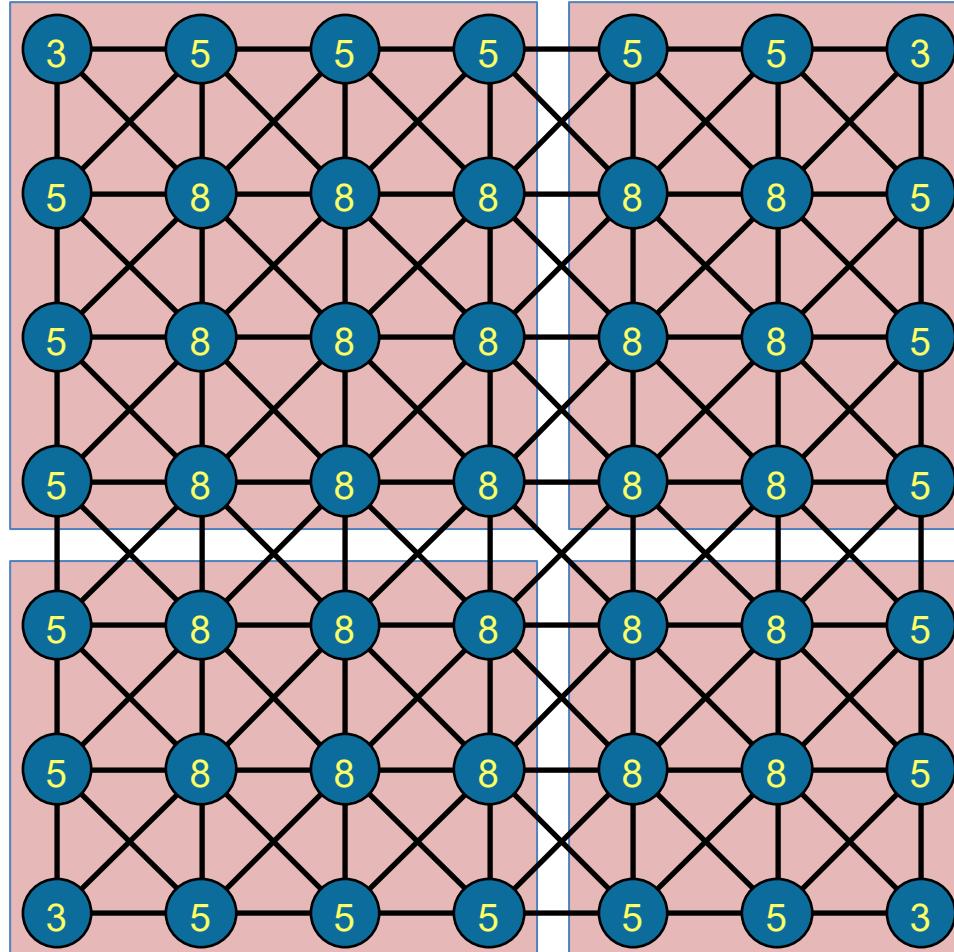
- AMG complexity growth for large 3-dimensional problems

$$\text{o.c.} = \frac{\sum_i \text{nnz}(A^{(i)})}{\text{nnz}(A)}$$



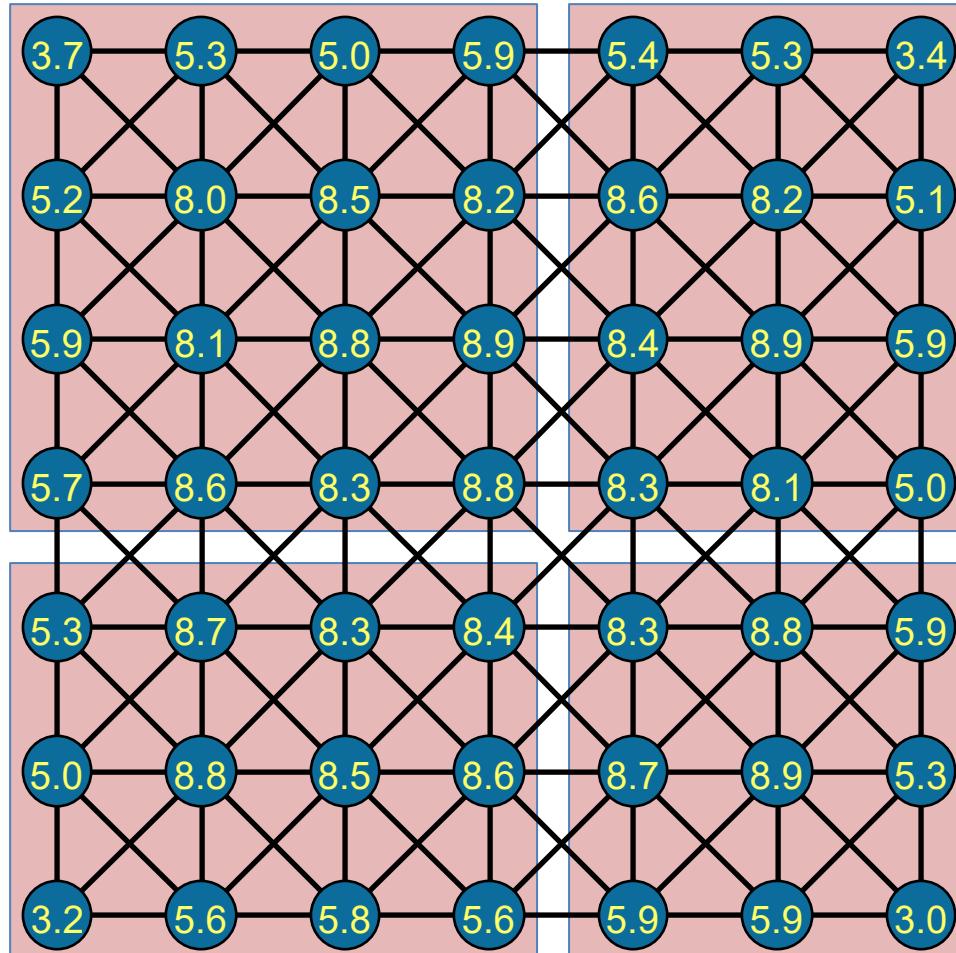
#procs	CLJP		
	#its	o.c.	time
1	9	5.6	11
8	9	6.5	21
64	11	6.7	38
128	10	7.9	57
256	11	7.8	76
512	11	7.2	128
1024	10	8.6	210

PMIS: start



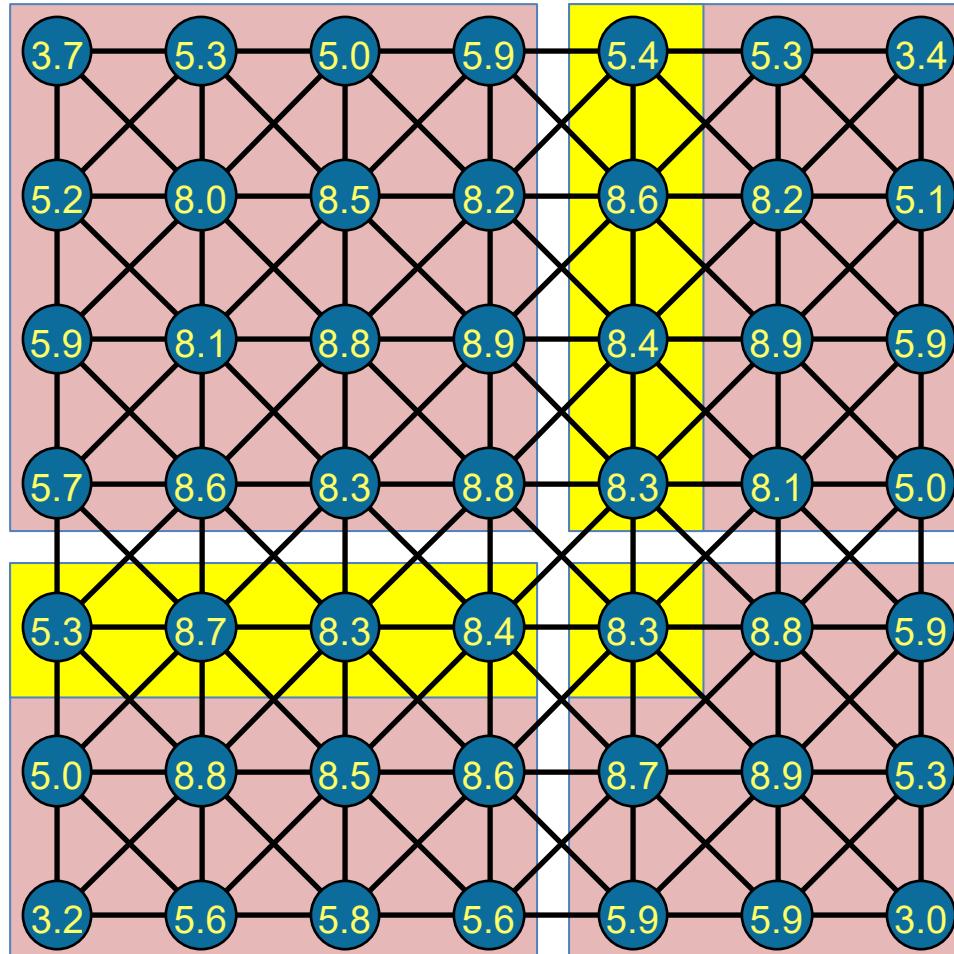
- select C-pt with maximal measure
- select neighbors as F-pts

PMIS: add random numbers



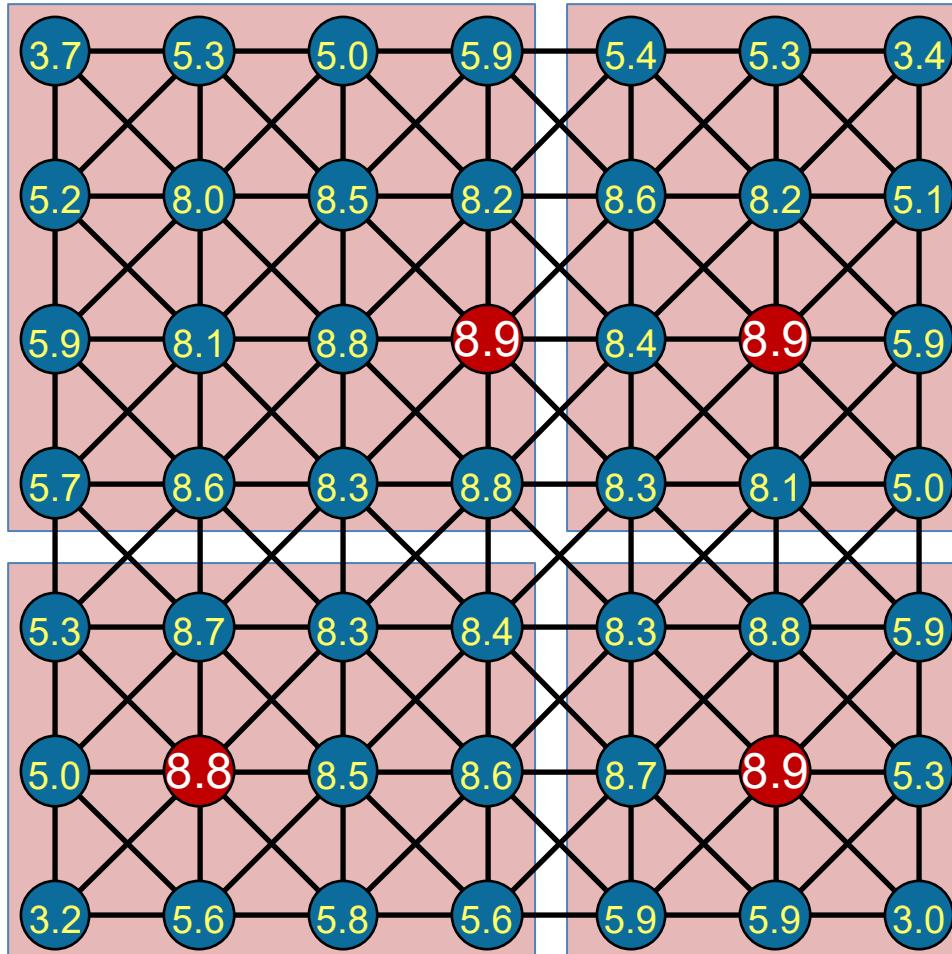
- select C-pt with maximal measure
- select neighbors as F-pts

PMIS: exchange neighbor information



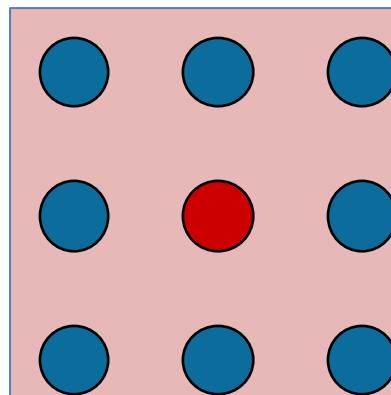
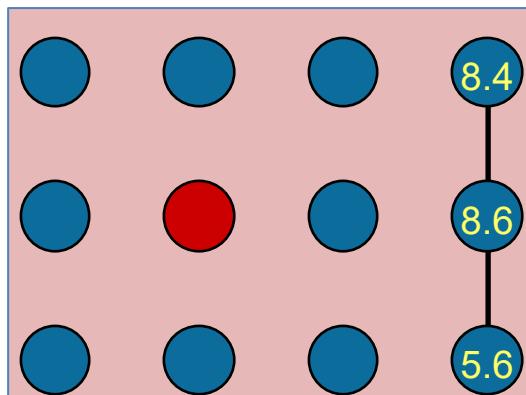
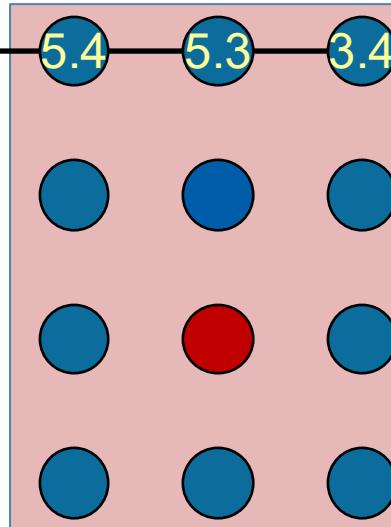
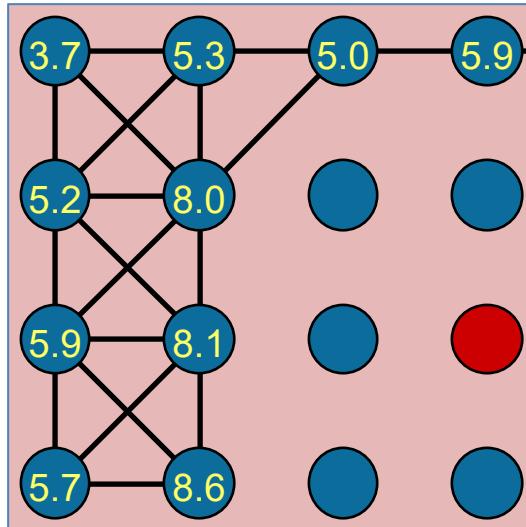
- select C-pt with maximal measure
- select neighbors as F-pts

PMIS: select



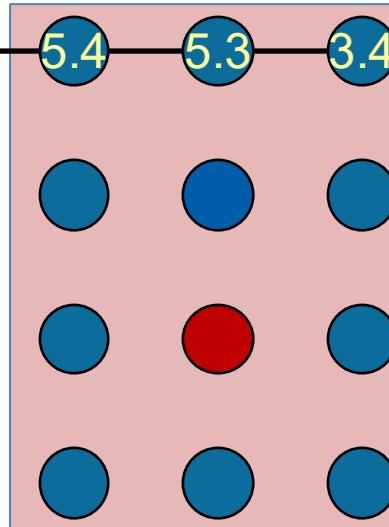
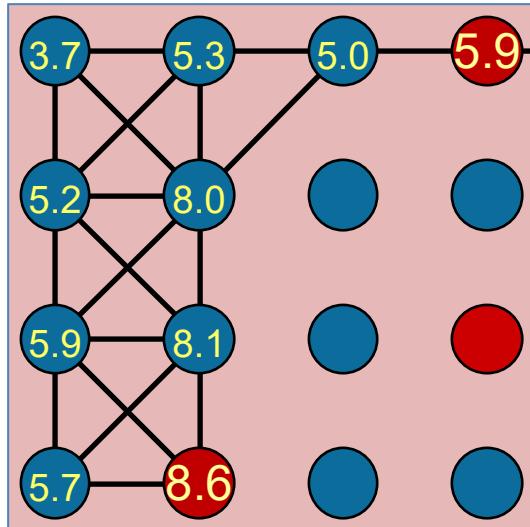
- select C-pts with maximal measure locally
- make neighbors F-pts

PMIS: update 1

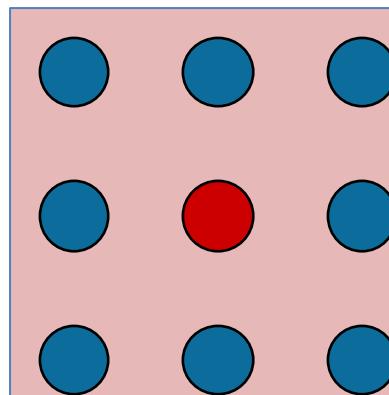
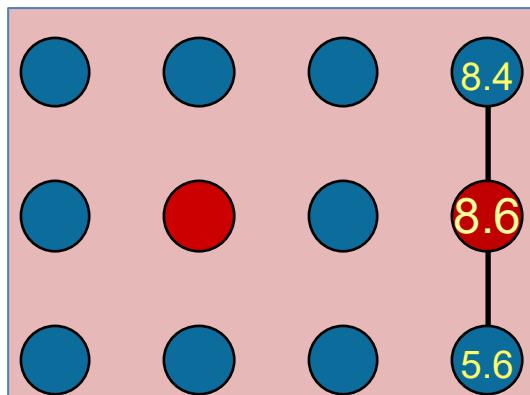


- select C-pts with maximal measure locally
- make neighbors F-pts
(requires neighbor info)

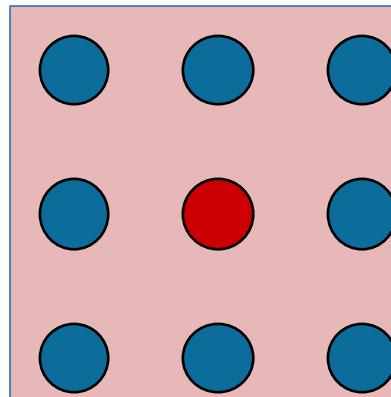
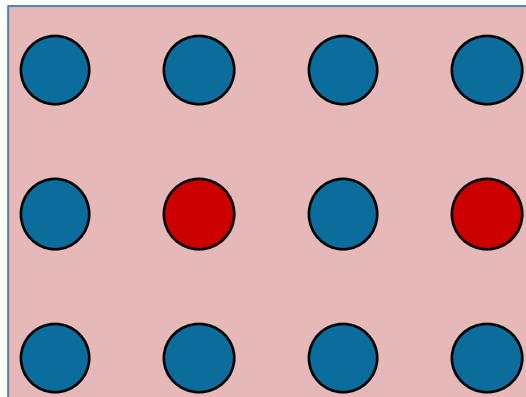
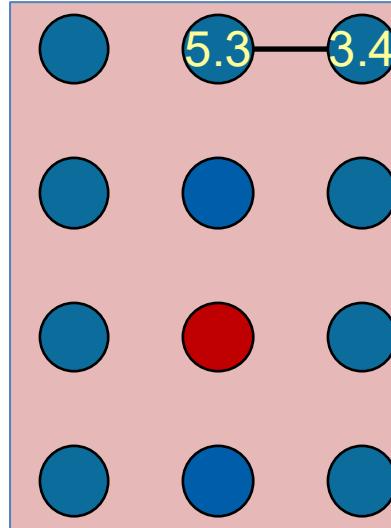
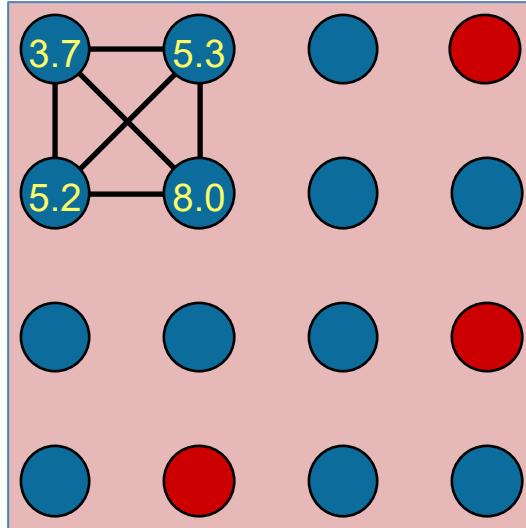
PMIS: select



- select C-pts with maximal measure locally
- make neighbors F-pts

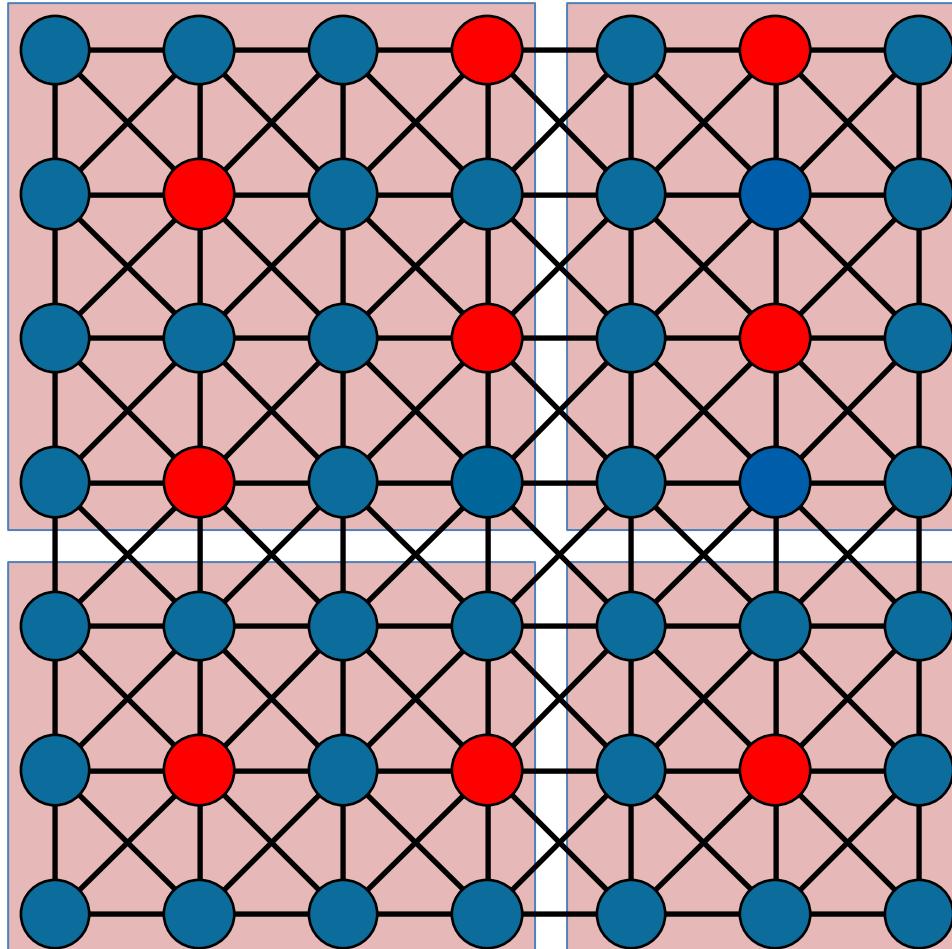


PMIS: update 2



- select C-pts with maximal measure locally
- make neighbors F-pts

PMIS: final grid



- select C-pts with maximal measure locally
- make neighbor F-pts
- remove neighbor edges

Unstructured Diffusion Problem with Jumps

- Leads to significantly lower complexities and better times, but decreases numerical scalability

#procs	CLJP			PMIS		
	#its	o.c.	time	#its	o.c.	time
1	9	5.6	11	18	1.5	7
8	9	6.5	21	30	1.6	16
64	11	6.7	38	62	1.5	29
128	10	7.9	57	64	1.5	31
256	11	7.8	76	72	1.5	27
512	11	7.2	128	118	1.6	55
1024	10	8.6	210	162	1.5	77

Parallel Coarsening Algorithms

- Several parallel algorithms (in *hypre*):
 - C-AMG variants
 - CLJP (Cleary-Luby-Jones-Plassmann) – one-pass approach with random numbers to get concurrency (illustrated next)
 - Falgout – C-AMG on processor interior, then CLJP to finish
 - PMIS – CLJP without the ‘C’; parallel version of C-AMG first pass
 - HMIS – C-AMG on processor interior, then PMIS to finish
 - CGC (Griebel, Metsch, Schweitzer) – compute several coarse grids on each processor, then solve a global graph problem to select the grids with the best “fit”
 - Aggressive coarsening to further decrease complexities – performs any of the above coarsening algorithms twice
- Other parallel AMG codes use similar approaches

Definitions and Notations

- Define: i strongly depends on j (and j strongly influences i) if $-a_{ij} \geq \theta \max_{k \neq i} (-a_{kj})$, $0 < \theta \leq 1$, with strong threshold θ
- Notations:
 - $N_i = \{j \mid a_{ij} \neq 0\}$
 - $S_i = \{j \in N_i \mid j \text{ strongly influences } i\}$
 - $F_i^s = F_i \cap S_i$
 - $C_i^s = C_i \cap S_i$
 - $N_i^w = N_i \setminus (F_i^s \cup C_i^s)$

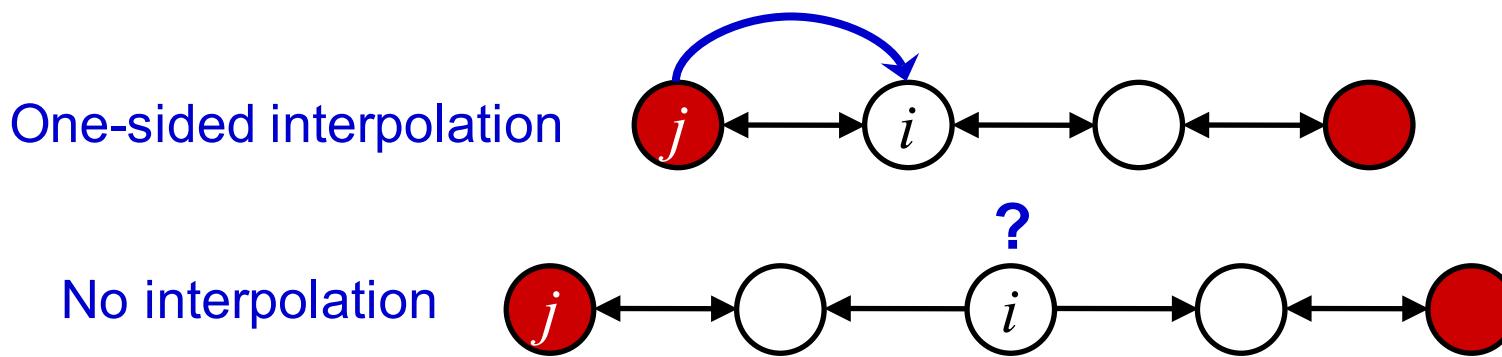
AMG Interpolation

- After relaxation: $Ae \approx 0$ (relative to e)
- Heuristic: error after interpolation should also satisfy this relation approximately
- Interpolation formula should satisfy $a_{ii} e_i \approx -\sum a_{ij} e_j$
- Direct interpolation: $w_{ik} = -\frac{\sum_{j \in N_i} a_{ij}}{\sum_{m \in C_i^S} a_{im}} \cdot \frac{a_{ik}}{a_{ii}}, \quad k \in C_i^S$
- Classical: Approximate error in strong F-connections, with weighted sum of common C-neighbors

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in N_i^W} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^S} \frac{a_{ik} \bar{a}_{kj}}{\sum_{m \in C_i^S} \bar{a}_{km}} \right), \quad j \in C_i^S$$

C-AMG interpolation is not suitable for more aggressive coarsening

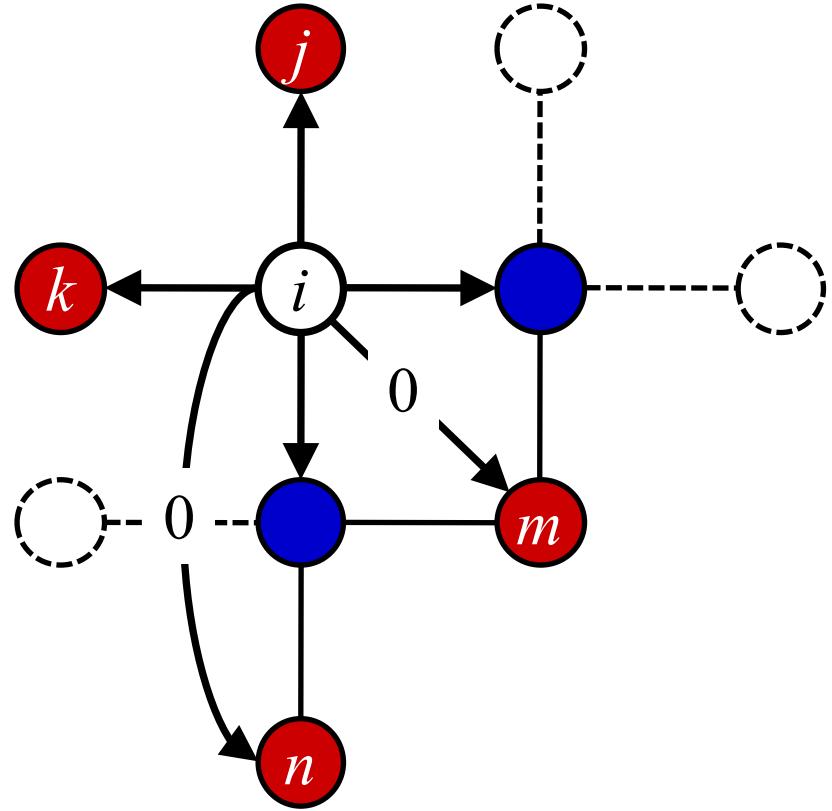
- PMIS is parallel and eliminates the second pass, which can lead to the following scenarios:



- Want above i -points to interpolate from both C -points
- Long-range (distance two) interpolation!

One possibility for long-range interpolation is extended interpolation

- C-AMG: $C_i = \{j, k\}$
- Long-range: $C_i = \{j, k, m, n\}$
- Extended interpolation – apply C-AMG interpolation to an extended stencil
- Extended+ i interpolation is the same as extended, but also collapses to point i
- Improves overall quality
- Note that this requires a 2nd layer of communication!



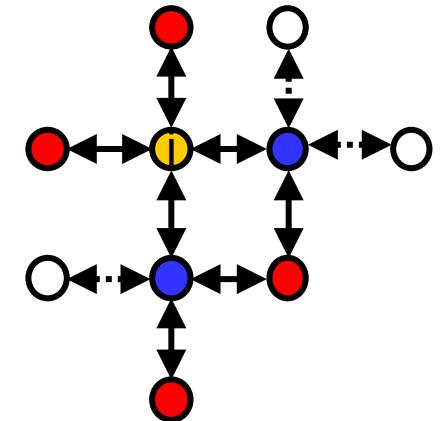
Distance-two interpolation operators

- Apply classical interpolation to extended stencil, \hat{C}_i^s
- Extended interpolation:

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in N_i^W - \hat{C}_i^s} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik} \bar{a}_{kj}}{\sum_{m \in \hat{C}_i^s} \bar{a}_{km}} \right),$$

$$j \in \hat{C}_i^s$$

$$\bar{a}_{ij} = \begin{cases} a_{ij}, & \text{if } a_{ij} a_{ii} > 0 \\ 0, & \text{otherwise} \end{cases}$$



- Extended+i interpolation: include a_{ki} ; add $\{i\}$ to \hat{C}_i^s

$$w_{ij} = -\frac{1}{\tilde{a}_{ii} + \sum_{k \in N_i^W - \hat{C}_i^s} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik} \bar{a}_{kj}}{\sum_{m \in \hat{C}_i^s \cup \{i\}} \bar{a}_{km}} \right),$$

$$\tilde{a}_{ii} = a_{ii} + \sum_{k \in F_i^s} \frac{a_{ik} \bar{a}_{ki}}{\sum_{m \in \hat{C}_i^s \cup \{i\}} \bar{a}_{km}}$$

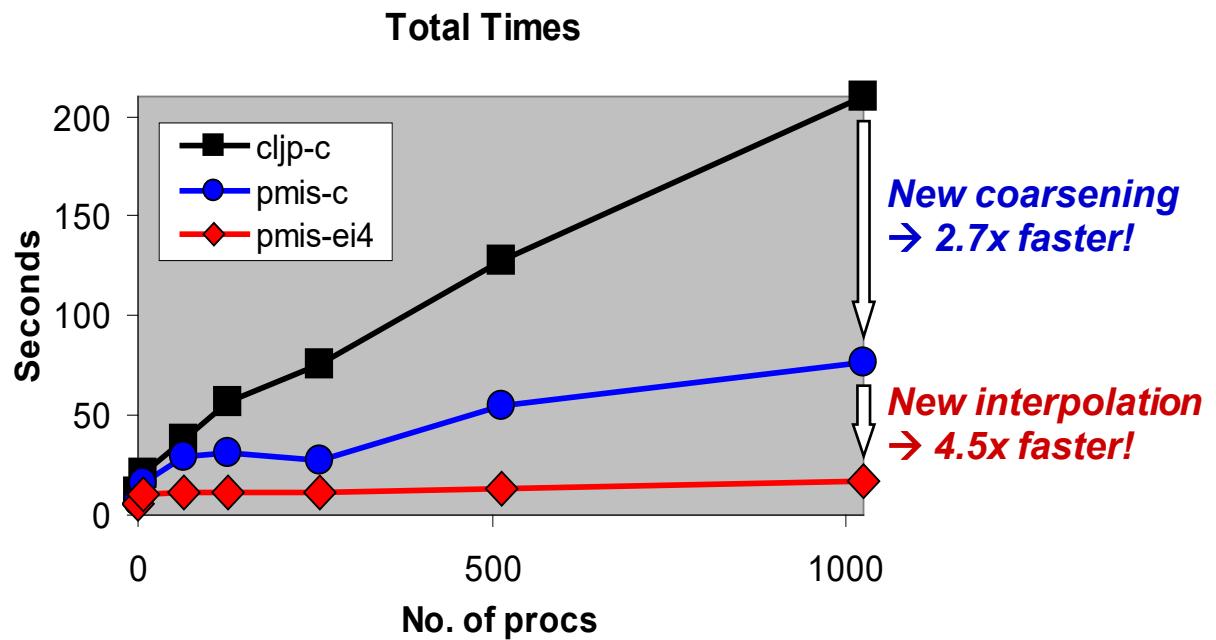
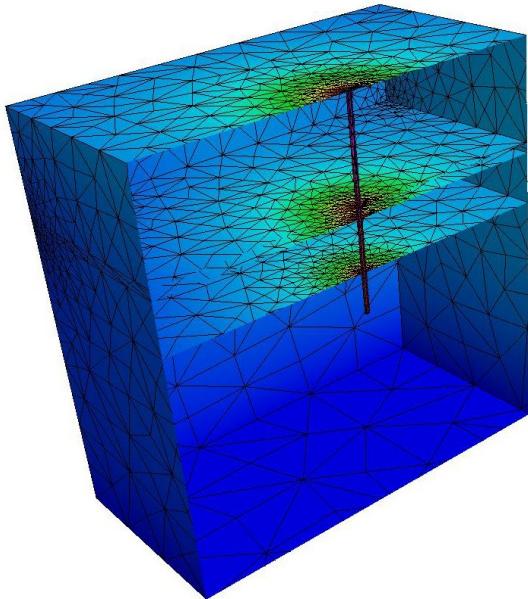
Unstructured Diffusion Problem with Jumps

#procs	CLJP		PMIS			
	class		class		e+i(4)	
	its	Op.c	its	Op.c	its	Op.c
1	9	5.6	18	1.5	9	1.8
8	9	6.5	30	1.6	11	1.9
64	11	6.7	62	1.5	13	1.9
128	10	7.9	64	1.5	12	1.9
256	11	7.8	72	1.5	13	1.8
512	11	7.2	118	1.6	12	1.8
1024	10	8.6	162	1.5	14	2.0

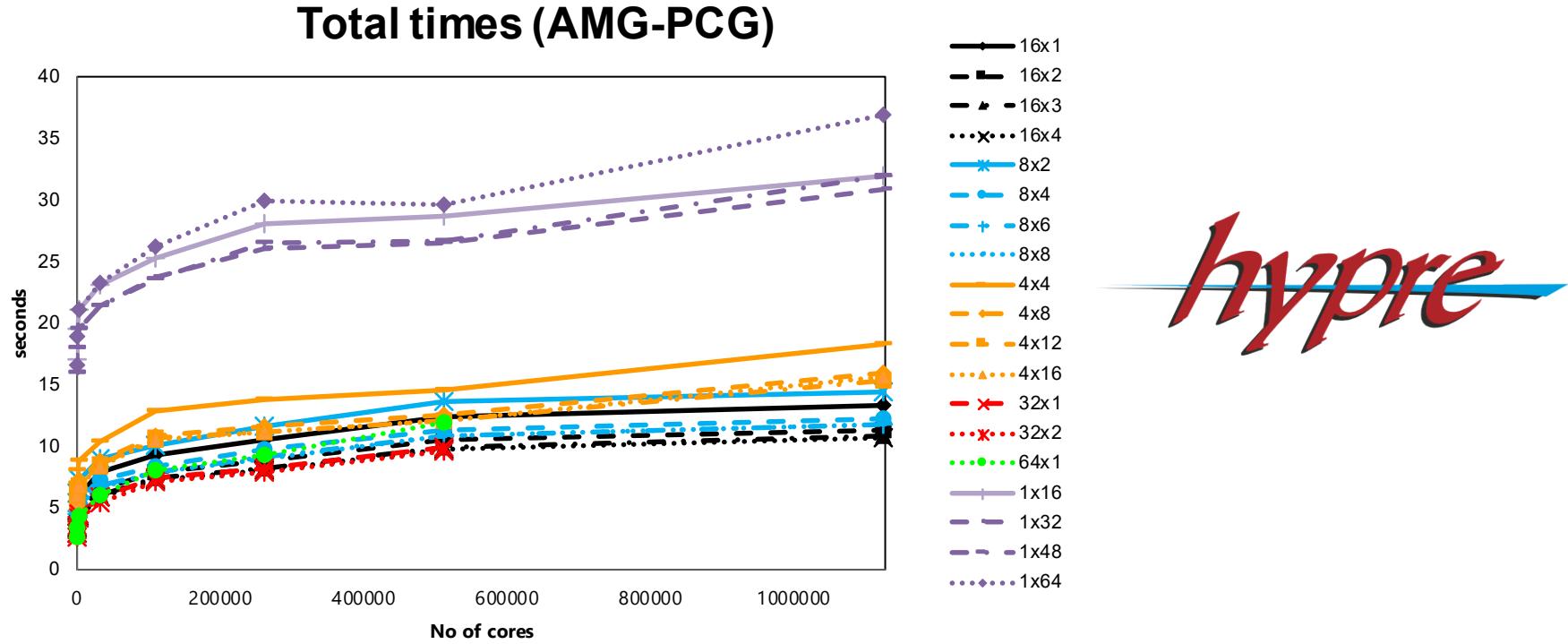
Note that full distance-two interpolation can still lead to large complexities and generally, is combined with interpolation truncation

New parallel coarsening and long-range interpolation methods improve scalability

- Unstructured 3D problem with material discontinuities
- About 90K unknowns per processor on MCR (Linux cluster)
- AMG - GMRES(10)



Parallel AMG in *hypre* scales to 1.1M cores on Sequoia (IBM BG/Q)



- $m \times n$ denotes m MPI tasks and n OpenMP threads per node
- Largest problem above: 72B unknowns on 1.1M cores
- Uses aggressive coarsening on first level (coarsen twice) followed by multipass interpolation

Interpolation operators suitable for GPUs

- While extended and extended+ i interpolation are parallel and can be efficiently implemented on parallel multicore machines suitable for larger grain parallelism, not suitable for GPUs
- Due to memory-efficient implementation and the need to distinguish fine and strong connections, there were many if statements, unsuitable for GPUs
- Suitable interpolation: direct interpolation, but: generally, not well convergent
- We designed a new class of interpolation operators, called MM-interpolation operators, that is similar but can be generated using basic matrix operations

MM-extended interpolation

- Consider $A = \begin{bmatrix} A_{FF} & A_{FC} \\ A_{CF} & A_{CC} \end{bmatrix}$ and $A = A^s + A^w$, define $P = \begin{bmatrix} W \\ I \end{bmatrix}$
- Generate A_{FF}^s and A_{FC}^s (requires communication)
- $D_\beta = \text{diag}(A_{FC}^s \mathbf{1})$; row sums of A_{FC}^s
- $D_\alpha = \text{diag}(\text{diag}(A_{FF}^s))$; diagonal of A_{FF}^s
- $D_w = \text{diag}([A_{FF}^w, A_{FC}^w] \mathbf{1})$; row sums of weak coefficients
- Then
$$W = -(D_\alpha + D_w)^{-1} (A_{FF}^s - D_\alpha + D_\beta) D_\beta^{-1} A_{FC}^s = -\tilde{A}_{FF}^s \tilde{A}_{FC}^s$$
- Final Matmat requires communication

New distance-two interpolation operators

- Extended interpolation:

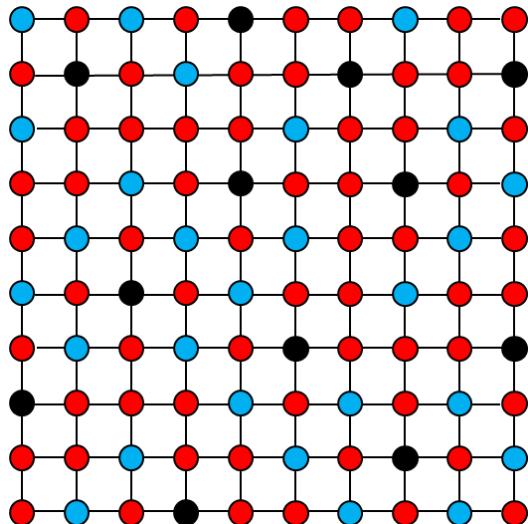
$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in N_i^W - \hat{C}_i^S} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^S} \frac{a_{ik} \bar{a}_{kj}}{\sum_{m \in \hat{C}_i^S} \bar{a}_{km}} \right), j \in \hat{C}_i^S$$

- MM-extended interpolation:

$$w_{ij} = -\frac{1}{a_{ii} + \sum_{k \in \textcolor{red}{N}_i^W} a_{ik}} \left(a_{ij} + \sum_{k \in F_i^S} \frac{a_{ik} \textcolor{red}{a}_{kj}}{\sum_{m \in \textcolor{red}{C}_k^S} \textcolor{red}{a}_{km}} \right), j \in \hat{C}_i^S$$

Other MM-interpolation operators

- MM-extended+i interpolation
- MM-extended+e interpolation
- Two-stage MM-interpolation operators for aggressive coarsening



C coarse points

C_x original C-points

F original fine points

$$P = \begin{bmatrix} I \\ Q^2 \\ Q^0 + Q^1 Q^2 \end{bmatrix} \quad \begin{array}{ll} Q^2 & : C_x \rightarrow C \\ Q^0 & : F \rightarrow C \\ Q^1 & : F \rightarrow C_x \end{array}$$

- For more details, see Ruipeng Li's talk on Thursday (Session 10A)

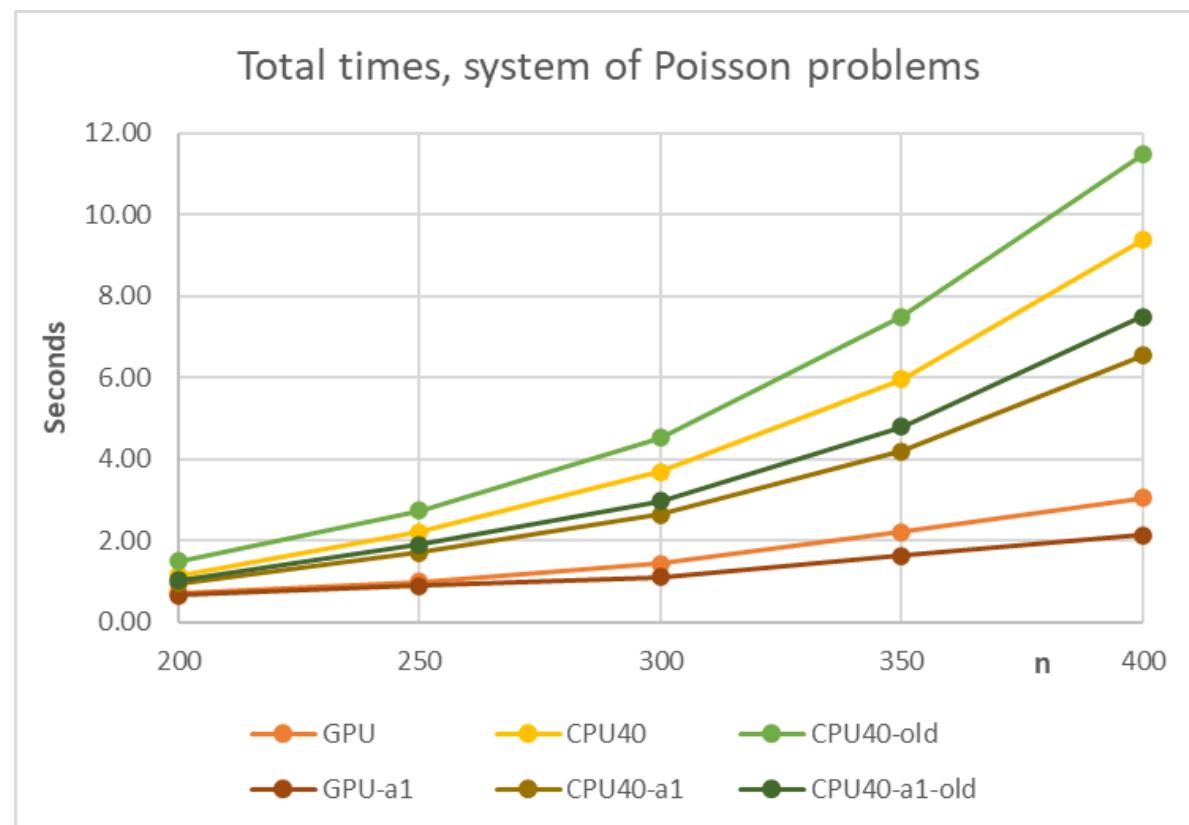
Comments on GPU implementation of AMG Setup Phase

- **Select coarse “grids”**
- **Define interpolation:** $P^{(m)}$, $m = 1, 2, \dots$
- **Define restriction:** $R^{(m)}$, $m = 1, 2, \dots$, often $R^{(m)} = (P^{(m)})^T$
- **Define coarse-grid operators:** $A^{(m+1)} = R^{(m)} A^{(m)} P^{(m)}$

- Implement generation of strength matrix (highly parallel)
- Implement fine-grained parallel coarsening algorithm: PMIS
- Implement interpolation based on matrix operations
- Implement coarse-grid operator generation – triple matrix product
- Can use efficient GPU kernels for much of this

Comparison CPU/GPU results on 16 nodes of Lassen (64 Nvidia GPUs, 640 Power 9 cores)

- Coupled 3D Poisson problem with 3 variables on a grid of size $n \times n \times n$, system size: $3n^3$
- a1: uses 1 level of aggr. coarsening
- GPU, CPU40: MM-ext+i
- GPU-a1, CPU40-a1: 2stage MM-ext+e
- CPU40-old: ext+i
- CPU40-a1-old: multipass



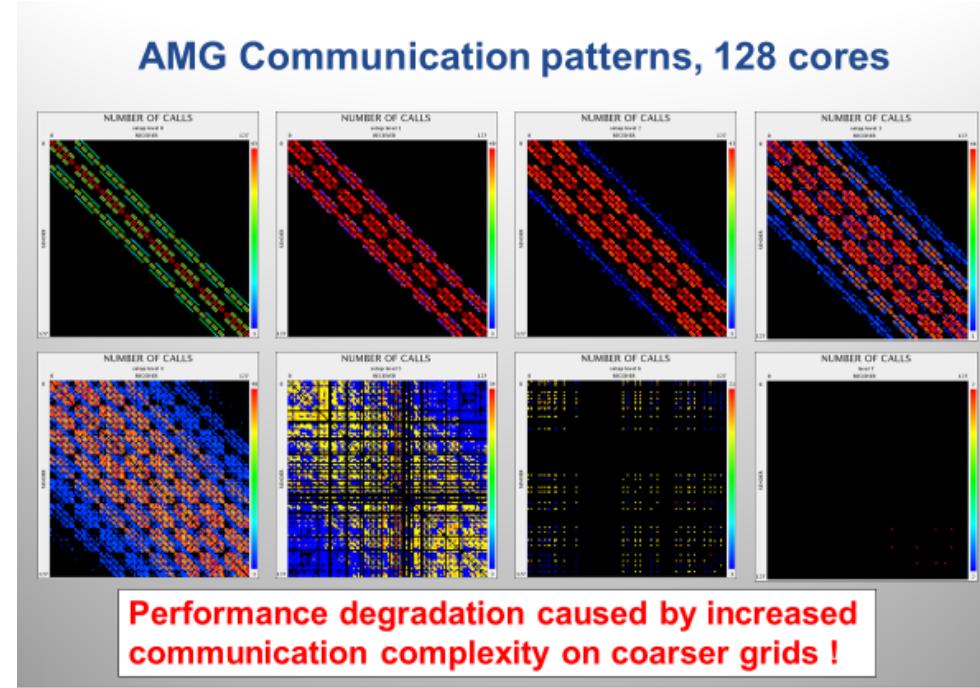
Parallelizing the Galerkin Product RAP

$$\begin{matrix} \textcolor{red}{RAP} \\ \left[\begin{array}{c} \textcolor{cyan}{\square} \\ \textcolor{blue}{\square} \end{array} \right] \end{matrix} = \begin{matrix} \textcolor{red}{R} \\ \left[\begin{array}{c} \textcolor{blue}{\square} \end{array} \right] \end{matrix} \quad \begin{matrix} \textcolor{red}{A} \\ \left[\begin{array}{c} \textcolor{blue}{\square} \end{array} \right] \end{matrix} \quad \begin{matrix} \textcolor{red}{P} \\ \left[\begin{array}{c} \textcolor{cyan}{\square} \\ \textcolor{blue}{\square} \end{array} \right] \end{matrix}$$

- Construction of coarse-grid operators by Galerkin method requires:
 - one layer of processor boundary data for P
 - one additional communication phase to add RAP contributions from other processors

Complexity Issues caused by Galerkin Product

- Coarse-grid selection in AMG can produce unwanted side effects
- Operator (RAP) “stencil growth” reduces efficiency
- For AMG we will therefore also consider complexities:
 - Operator complexity:
 $C_{op} = (\sum_{i=0}^L nnz(A_i))/nnz(A_0)$
 - Affects flops and memory
 - Generally, would like $C_{op} < 2$, close to 1
- Can control complexities in various ways
 - varying strength threshold
 - more aggressive coarsening
 - Operator sparsification (interpolation truncation, non-Galerkin approach)
- Needs to be done carefully to avoid excessive convergence deterioration

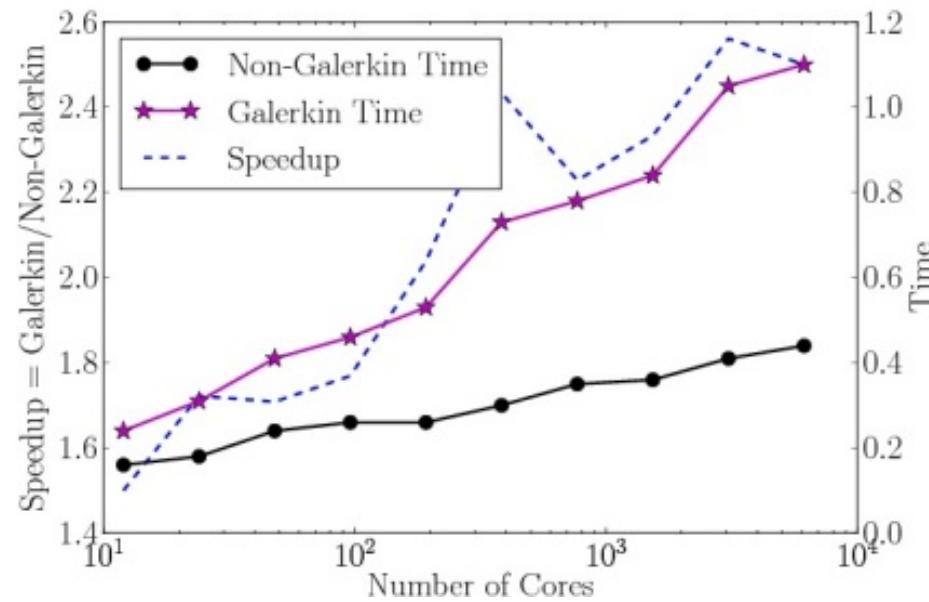


Efforts to Reduce Communication

- Reduction of number of nonzeros per row in Galerkin product
- Use of interpolation truncation (was included in results shown earlier)
- Sparsification of the coarse grid operator
- Agglomeration of coarsest grid
- AMG-DD

Reducing parallel communication costs (1)

- Non-Galerkin AMG replaces the usual coarse-grid operators with sparser ones
 - Speedups from 1.2x - 2.4x over existing AMG
 - In *hypre* 2.10.0b

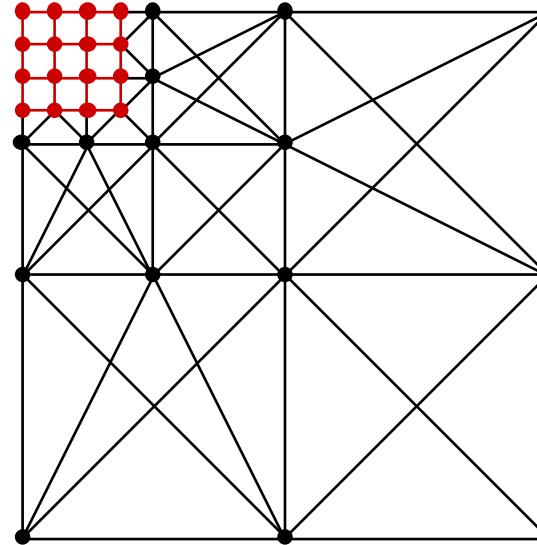


Reducing parallel communication costs (2)

- One technique that can be useful is to gather (agglomerate) the system matrix at some coarse level to a single processor
- This costs $O(\log P)$ communications, but so does simply continuing the V-cycle
- This helps when the matrix rows have grown in complexity
 - Avoids cost of communicating with many neighbors

Reducing parallel communication costs (3)

- AMG domain decomposition (AMG-DD) employs cheap global problems to speed up convergence
 - Constructs problems algebraically from an existing method
 - Potential for FMG convergence with only $\log N$ latency (vs $\log^2 N$)!
 - Implemented parallel code

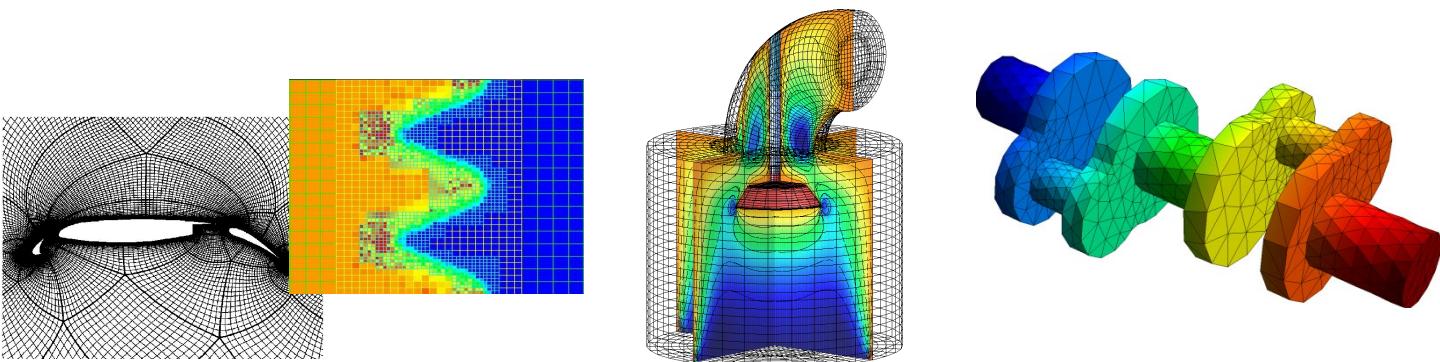


- For details and much newer info see Wayne Mitchell's talk on Monday, Session 5B

Multigrid Software Design

Simulation codes present a wide array of challenges for scalable linear solver libraries

- Different **applications**
 - Diffusion, elasticity, magnetohydrodynamics (MHD)
- Different **discretizations** and **meshes**
 - Structured, block-structured, structured AMR, overset, unstructured



- Different **languages** – C, C++, Fortran
- Different **programming models** – MPI, OpenMP
- **Scalability beyond 100,000 processors!**

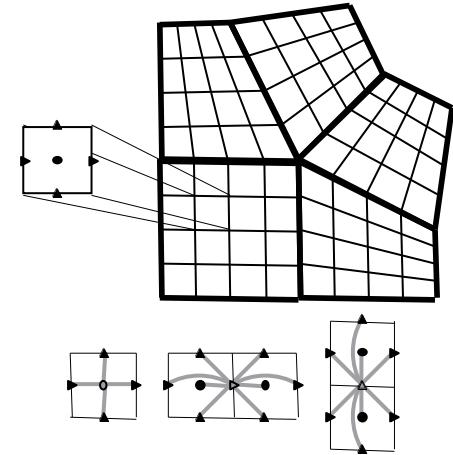
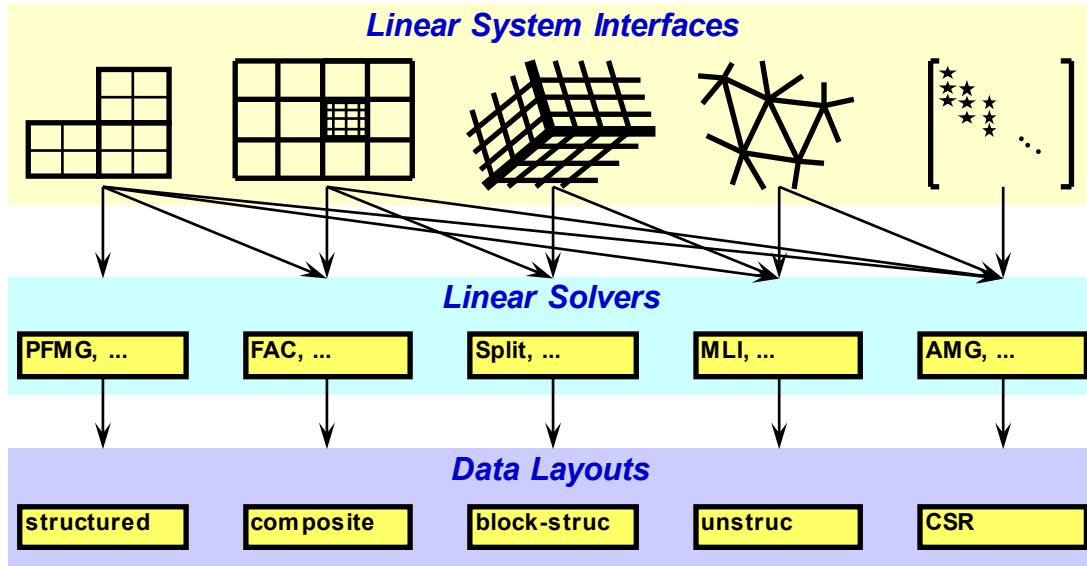
Challenge: Software design

- The “best” solver for a given application usually takes advantage of the setting
 - structured grids, constant coefficients, FE discretization, etc.
- Traditional linear solver libraries take in only generic matrix-vector information
- How do we supply these “best solvers” in library form?

Some available multigrid software

- ML, MueLu included in 
- GAMG in 
- The ~~hypre~~ library provides various algebraic multigrid solvers, including multigrid solvers for special problems e.g., Maxwell equations, ...
- All of these provide different flavors of multigrid and provide excellent performance for suitable problems
- Focus here on 

Unique software interfaces in *hypre* provide efficient solvers not available elsewhere



Block-structured grid with 3 variable types and 3 discretization stencils

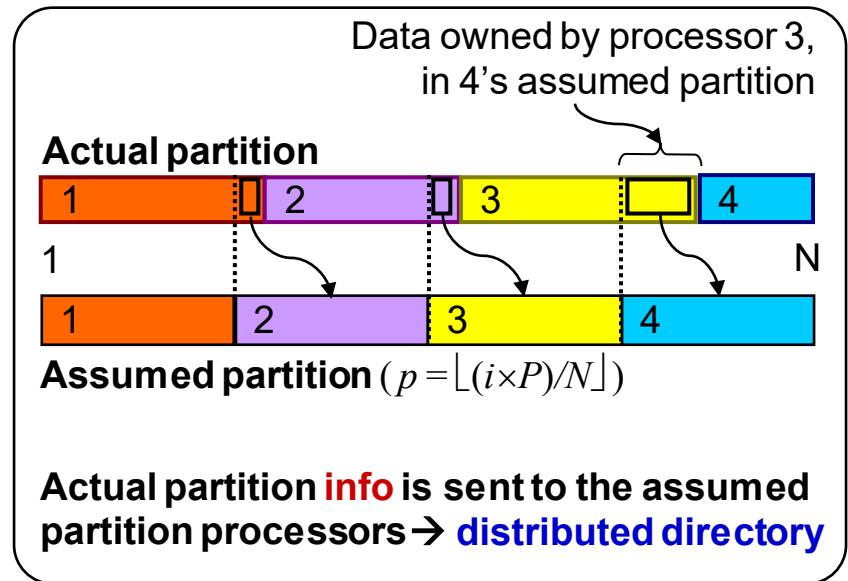
- Example: *hypre*'s interface for semi-structured grids
 - Based on “grids” and either “stencils” or “finite elements” (**new**)
 - Allows for **specialized solvers** for structured AMR
 - Also provides for more **general solvers** like *AMG*

Challenge: Parallel implementation

- Simple algorithms can be used for modest numbers of processors (< 100)
 - e.g., store $O(P)$ data and do $O(P^2)$ computations to determine send/receive patterns
- On large numbers of processors (1K – 10K), algorithms get more complex
 - e.g., store $O(P)$ data and do $O(P)$ computations to determine send/receive patterns
- What about 100K – 1B processors?

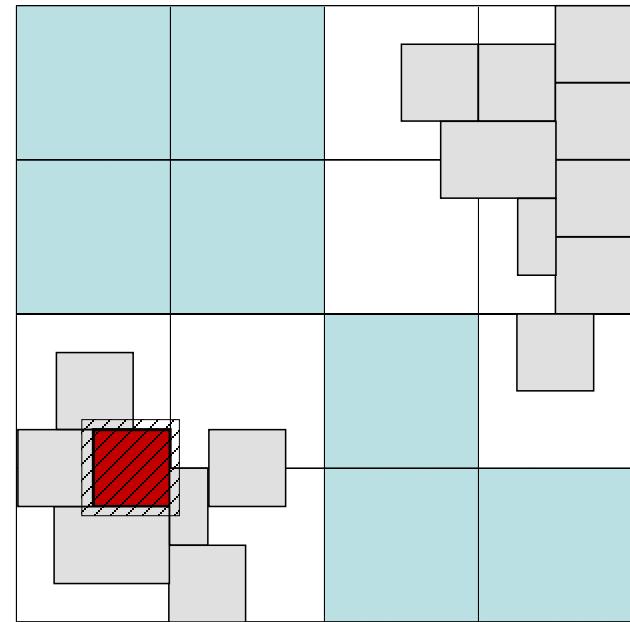
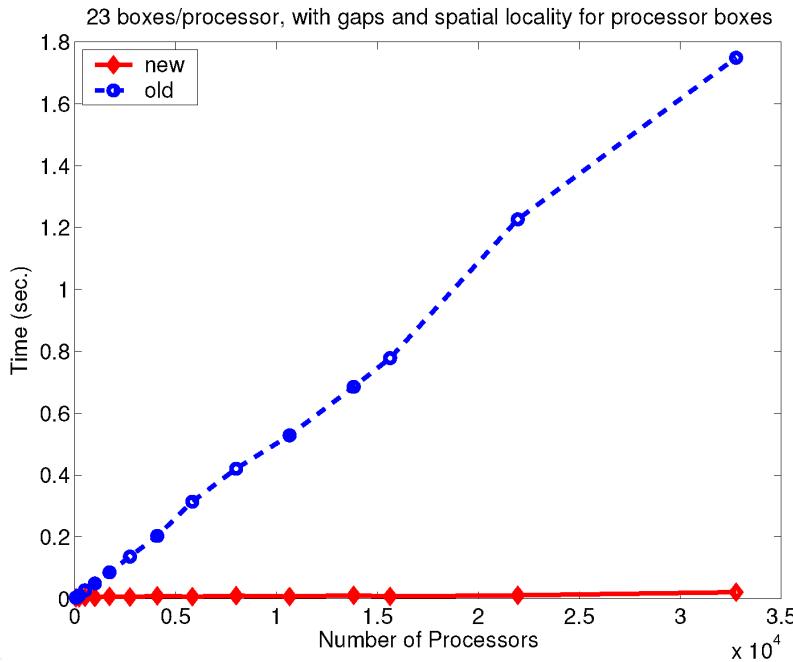
Assumed partition (AP) algorithm enables scaling to 100K+ processors

- Answering global distribution questions previously required $O(P)$ storage & computations
 - On BG/L, $O(P)$ storage may not be possible
-
- AP algorithm requires
 - $O(1)$ storage
 - $O(\log P)$ computations
 - Default approach in *hypre*
 - AP has general applicability beyond *hypre*



Assumed partition (AP) algorithm is more challenging for structured AMR grids

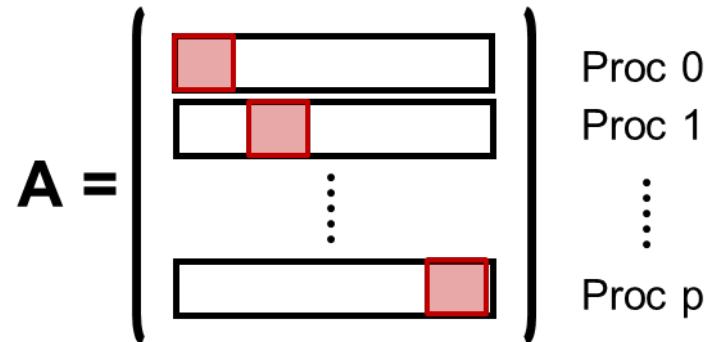
- AMR can produce grids with “gaps”
- Our AP function accounts for these gaps for scalability
- Demonstrated on 32K procs of BG/L



Simple, naïve AP function leaves processors with empty partitions

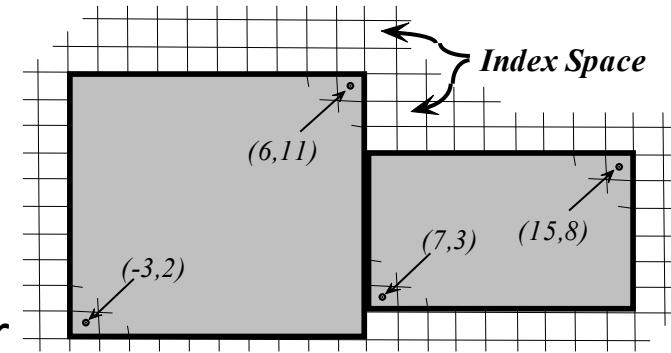
ParCSRMatrix data structure

- Based on compressed sparse row (CSR) data structure
- Consists of two CSR matrices:
 - One containing **local coefficients** connecting to local column indices
 - The other (Offd) containing coefficients with column indices pointing to off processor rows
- Also contains a mapping between local and global column indices for Offd
- Requires much indirect addressing, integer computations, and computations of relationships between processes etc,



Structured-Grid System Interface (Struct)

- Appropriate for scalar applications on structured grids with a fixed stencil pattern
- A lot of potential for GPUs!!
- Grids are described via a global d -dimensional *index space*, arrays of boxes, and a box manager
- A *box* is a collection of cell-centered indices, described by its “lower” and “upper” corners
- Matrices and vectors have a grid, ghost layer info, data space and data
- Matrices also have a stencil and a communication package

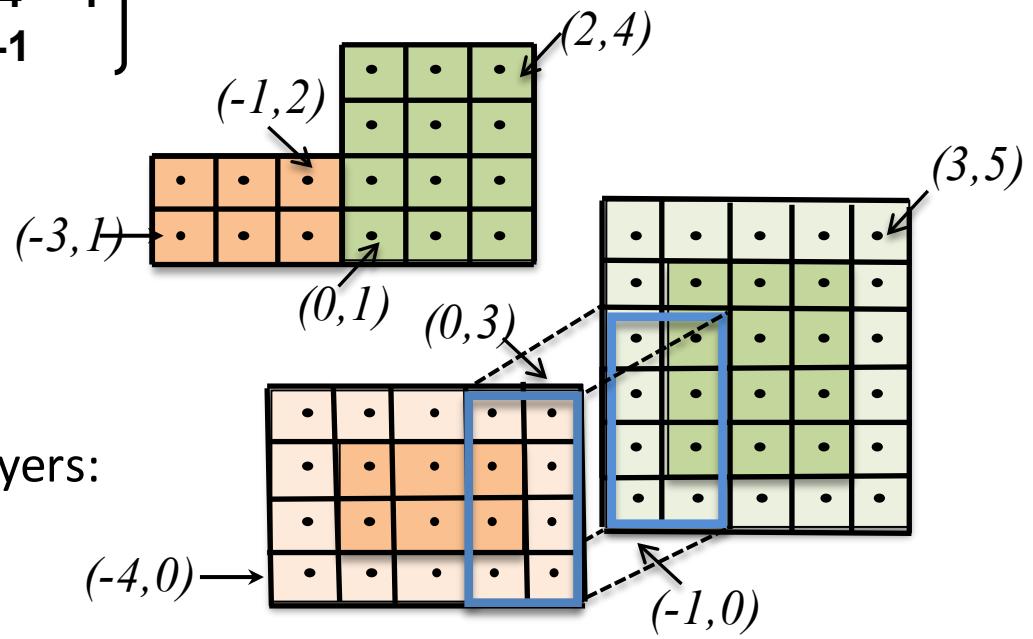


How is matrix data currently stored

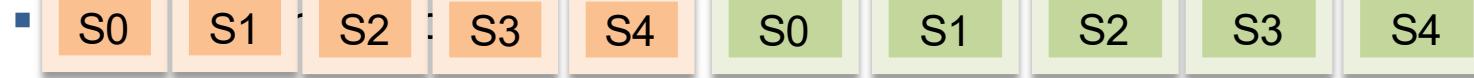
- Stencil

$$\begin{bmatrix} & S4 \\ S1 & S0 & S2 \\ & S3 \end{bmatrix} = \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

- Grid boxes: $[(-3,1), (-1,2)]$
 $[(0,1), (2,4)]$



- Data Space: grid boxes + ghost layers:
 $[(-4,0), (0,3)]$, $[(-1,0), (3,5)]$



BoxLoops

```
hypre_BoxLoop2Begin (ndim, loop_size,\  
                      dbox1, start1, stride1, i1,\  
                      dbox2, start2, stride2, i2)\\  
hypre_BoxLoop2For (i1, i2)  
{  
    x(i1) += y(i2);  
}  
hypre_BoxLoop2End (i1, i2);
```

ndim = 2

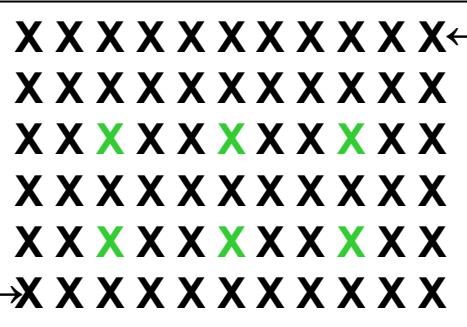
loop_size = (3,2)

start1 = (2,1)
stride1 = (3,2)

start2 = (2,2)
stride2 = (1,1)

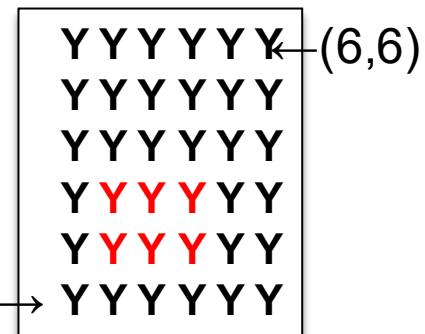
dbox1 = [(0,0),(10,5)]

(0,0)



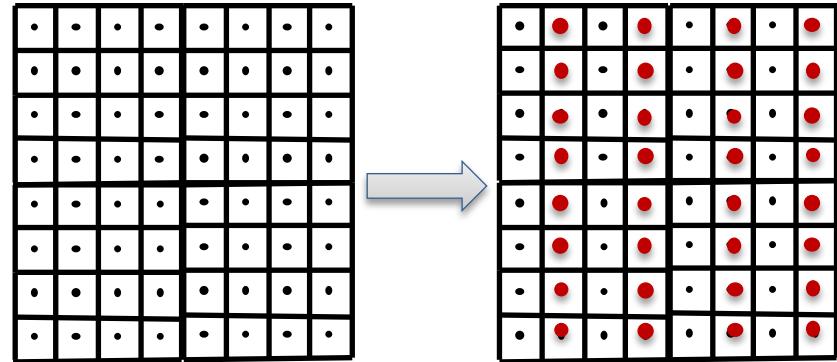
dbox2 = [(1,1),(6,6)]

(1,1)



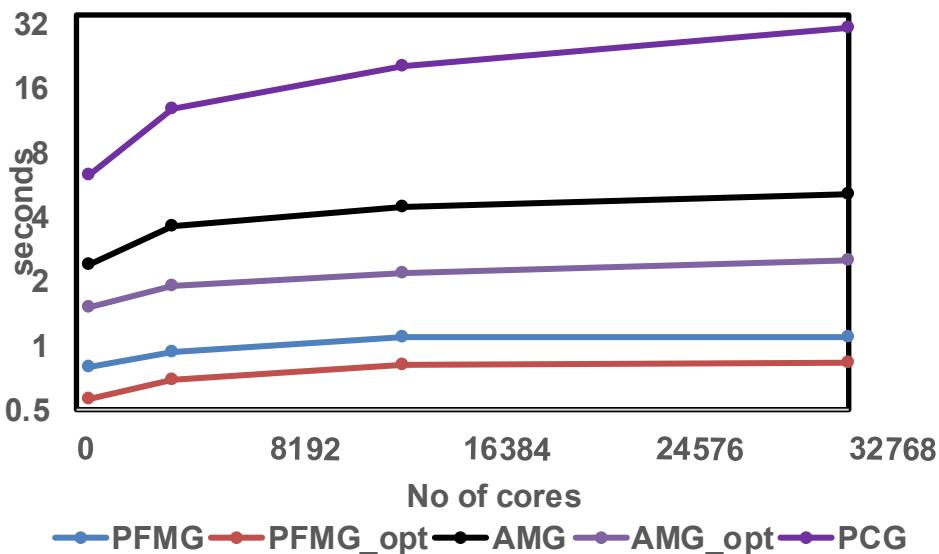
PFMG is an algebraic multigrid method for structured grids

- Matrix A defined in terms of grids and stencils
- Uses semicoarsening
- Simple 2-point interpolation P
→ limits stencil growth in coarse grid operator $P^T AP$ to at most 9pt (2D), 27pt (3D)
- Optional non-Galerkin approach (Ashby, Falgout), uses geometric knowledge, **preserves stencil size**
- Pointwise smoothing
- Highly efficient for suitable problems

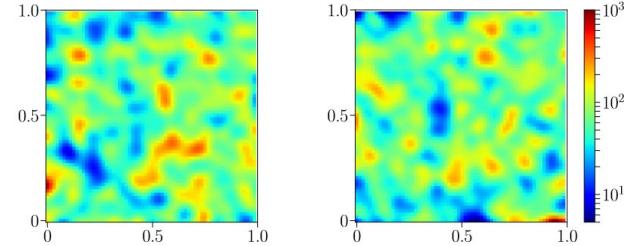


Algebraic multigrid as preconditioner

- Generally algebraic multigrid methods are used as preconditioners to Krylov methods, such as conjugate gradient (CG) or GMRES
- This often leads to additional performance improvements



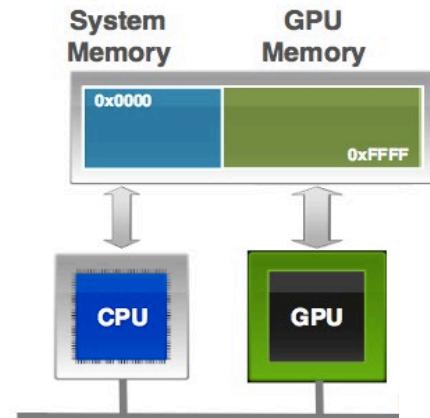
Classic porous media diffusion problem:
 $-\nabla \cdot \kappa \nabla u = f$
with κ having jumps of 2-3 orders of magnitude



Weak scaling: 32x32x32 grid points per core,
BG/Q

Efforts to move hypre to emerging architectures

- Different strategies for different interfaces
 - Unstructured matrix data structures (ParCSR, consists of 2 CSR matrices)
 - Structured and semi-structured matrix data structures
- Investigation of replacing routines or loops with code suitable to be run on GPUs
 - Parallelize using OpenMP4 pragmas
 - Replace basic kernels with Cuda functions
 - Use RAJA underneath
 - Use Kokkos underneath
- Need to consider where and how to store data
 - Explicit copies
 - Unified Memory



New memory model in hypre

- Define new memory model general enough to be useful for future heterogeneous architectures
- Three conceptual memory locations:
 - HYPRE_MEMORY_DEVICE
 - HYPRE_MEMORY_HOST
 - HYPRE_MEMORY_SHARED
- Mapped to different physical memory in different configurations

	HYPRE_MEMORY_HOST	HYPRE_MEMORY_DEVICE	HYPRE_MEMORY_SHARED
C1	HOST	HOST	HOST
C2	HOST	CUDADEVICE	CUDADEVICE
C3	HOST	CUDADEVICE	CUDAMANAGED

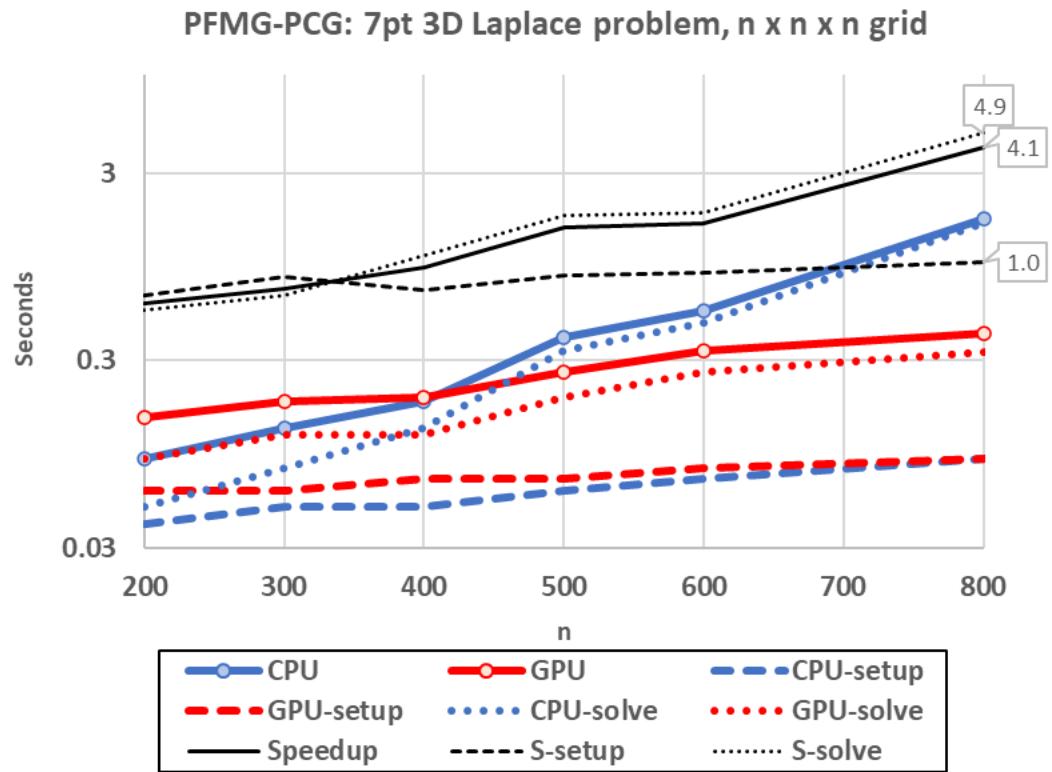
- C1: Non-GPU configuration
- C2: --with-cuda or –with-device-openmp
- C3: C2 + --enable-unified-memory

Strategy for interfaces/solvers for preparation for exascale platforms

- Strategy for structured solvers:
 - More suitable data structures (based on grids and stencils) allow easier porting through adapting of the BoxLoops
 - Currently allow use of CUDA, OMP4.5, RAJA, Kokkos
 - BoxLoops also ease porting to AMD and Intel GPUs
- Strategy for unstructured solvers, particularly for most often used AMG preconditioner/solver BoomerAMG
 - Modularize into smaller chunks/kernels to be ported to CUDA for Nvidia GPUS initially
 - Develop new algorithms for portions not suitable for GPUs (interpolation operators, smoothers)
 - Convert CUDA kernels to HIP for AMD GPUs and DPC++ for Intel GPUs (in progress)
 - Various special solvers (e.g., Maxwell solver AMS, ADS, pAIR, MGR) built on BoomerAMG and will benefit from this strategy
- Strategy for semi-structured interfaces/solvers:
 - Mostly built on structured interface with some unstructured parts

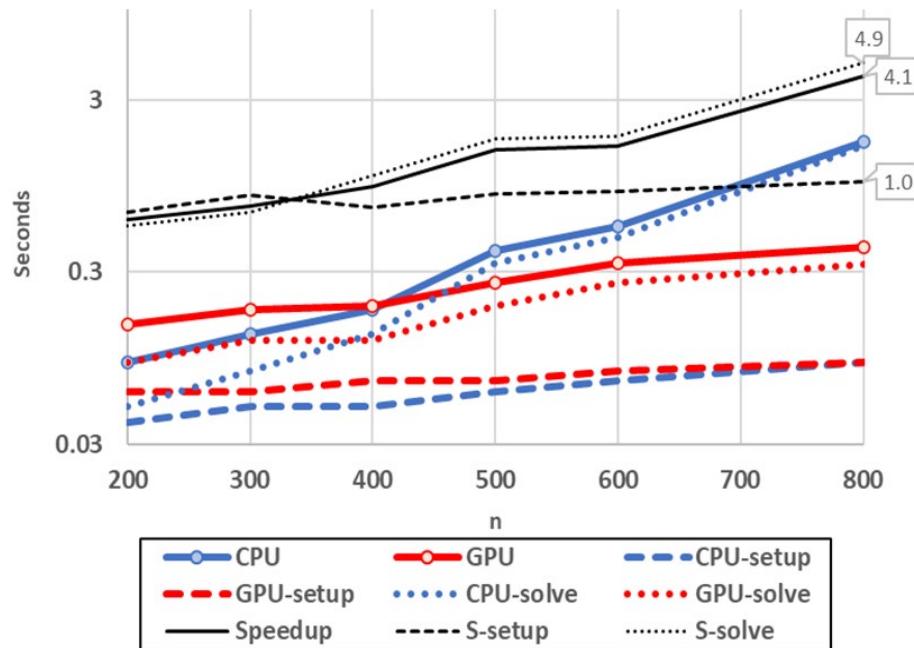
Results on Lassen: PFMG-PCG for 7pt 3D Laplace problem

- CPU, 4 MPI x 10 OpenMP / node (fastest CPU runs, 40 MPI/node is slower)
- GPU, 4 MPI / node
- Run on 16 nodes, 64 GPUs, from 0.5M/GPU to 8M/GPU
- Cross-over at 1M/GPU for total times and solve times
- Setup times on GPU generally slower, except for largest problem, but setup times are overall very small

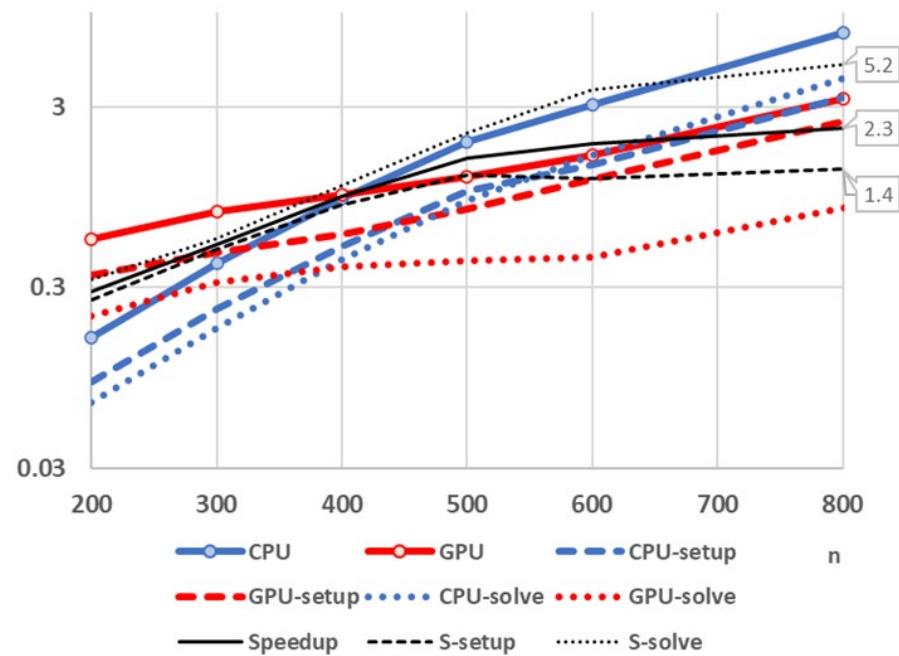


Results on Lassen: AMG-PCG for a 7pt 3D Laplace problem on a $n \times n \times n$ grid

PFMG-PCG: 7pt 3D Laplace problem, $n \times n \times n$ grid

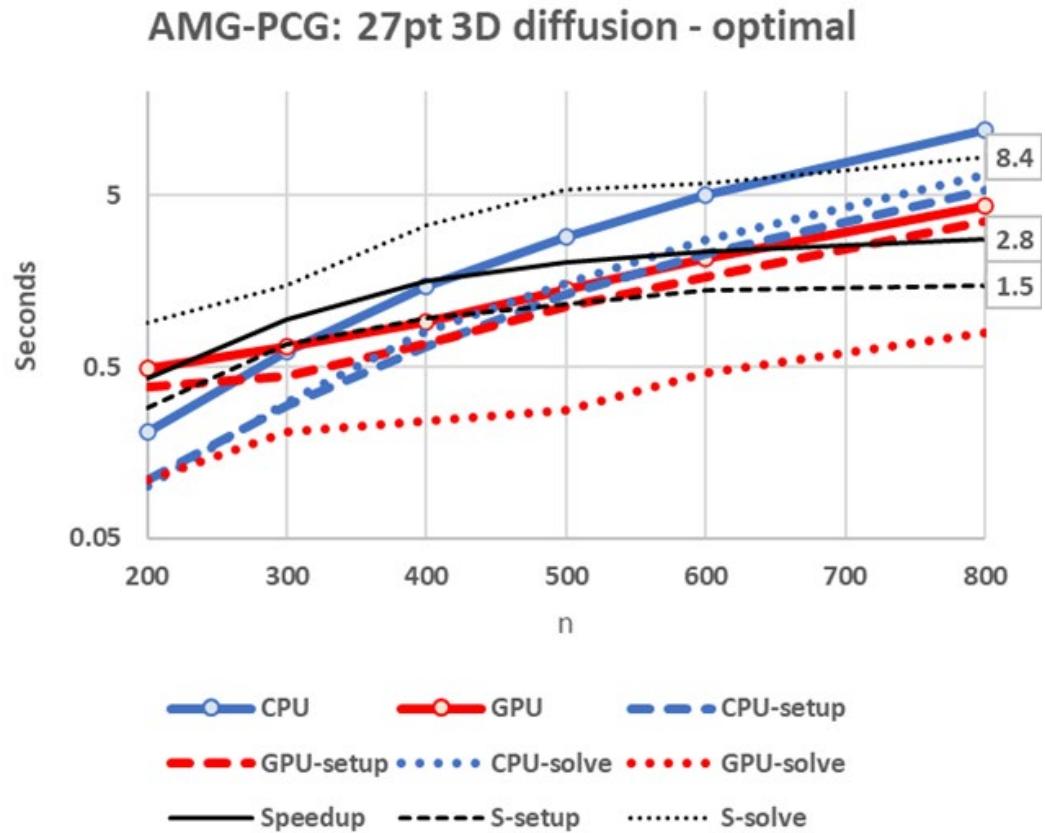


AMG-PCG: 7pt 3D Laplace problem, $n \times n \times n$ grid



Results on Lassen: AMG-PCG for a 27pt 3D diffusion problem on a $n \times n \times n$ grid

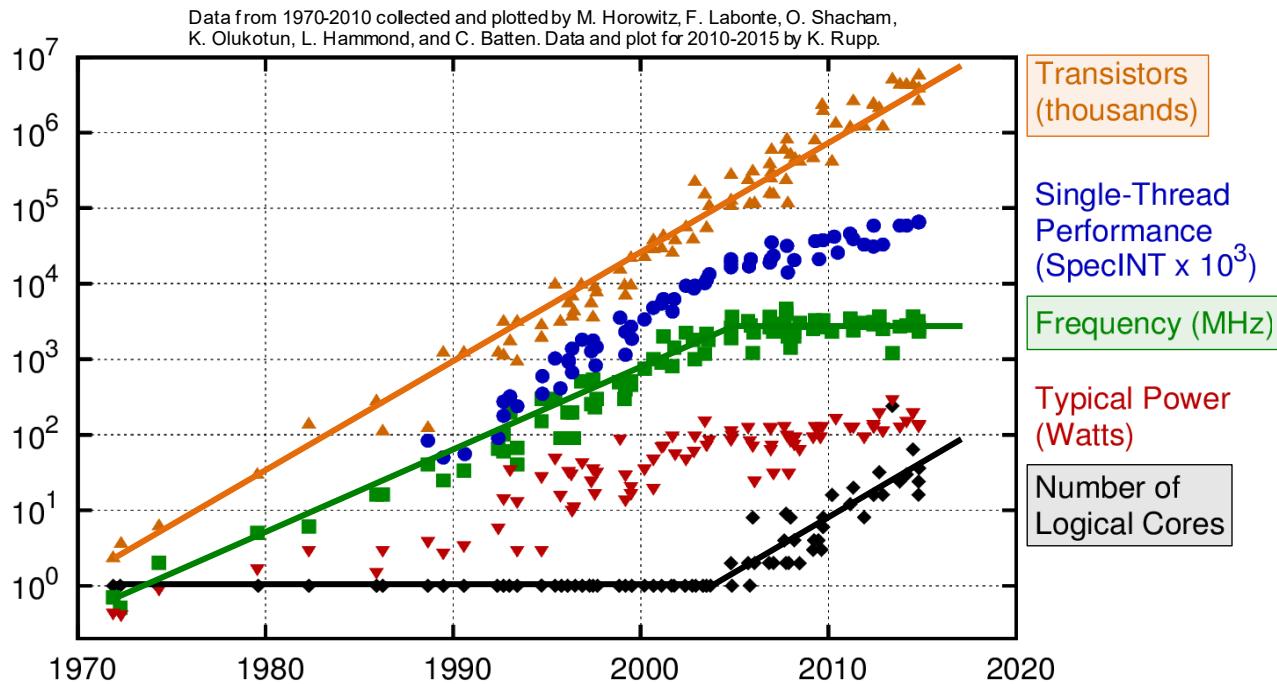
- CPU, 4 MPI x 10 OpenMP / node
- GPU, 4 MPI / node
- Run on 16 nodes, 64 GPUs, from 0.5M/GPU to 8M/GPU
- Cross-over at <.5M/GPU for total times, 1M/GPU for setup times



Parallel Time Integration (Parallel Multigrid in Time)

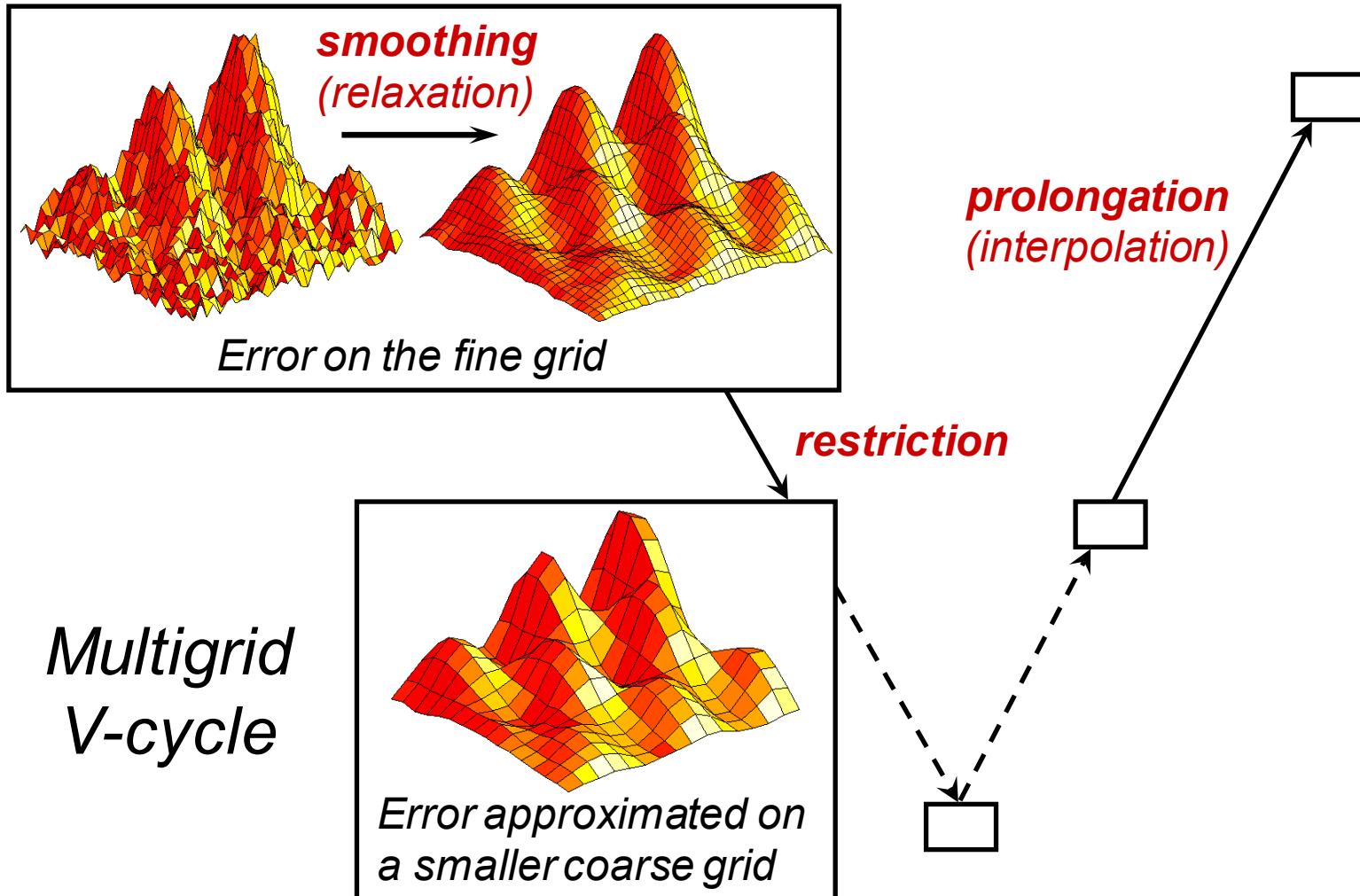


Parallel time integration is a major paradigm shift driven by hardware design realities



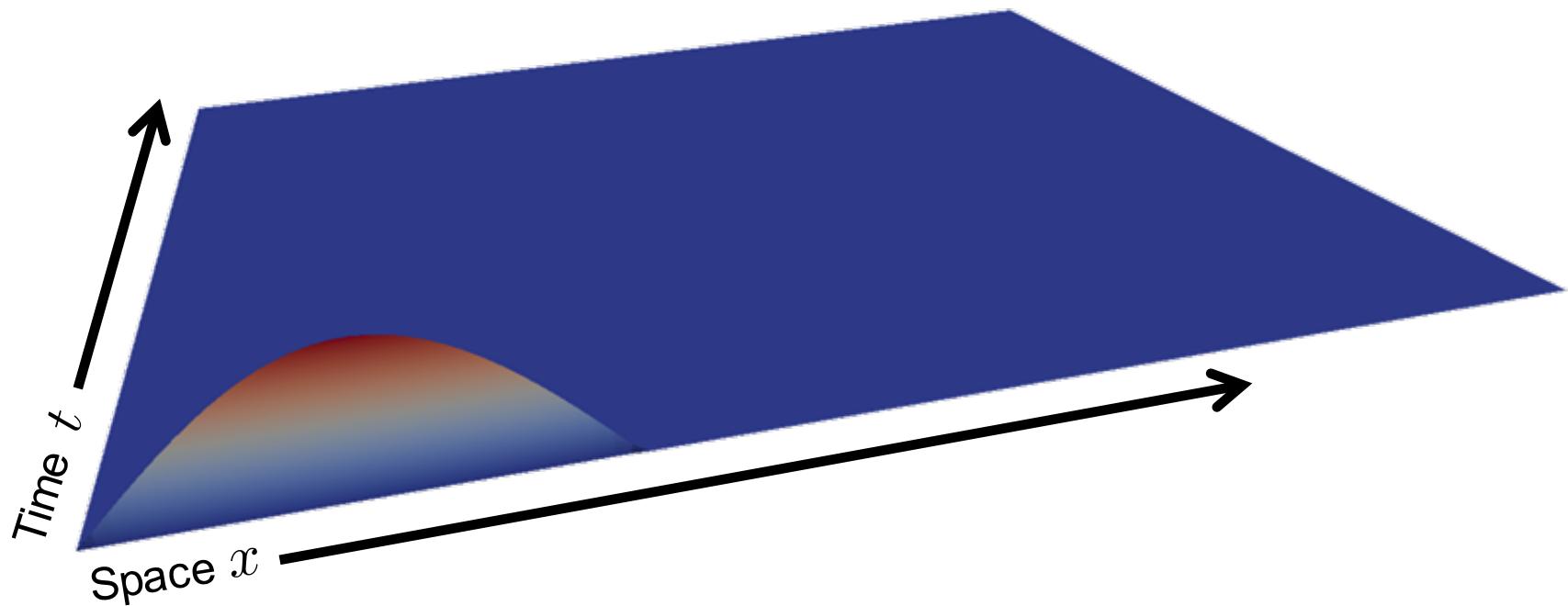
- Architecture trend: flat clock rates, more concurrency
 - Traditional time stepping is becoming a sequential bottleneck
- Continued advancement in scientific simulation will require algorithms that are parallel in time

One approach for parallel-in-time: apply multigrid ideas to the (space-)time dimension



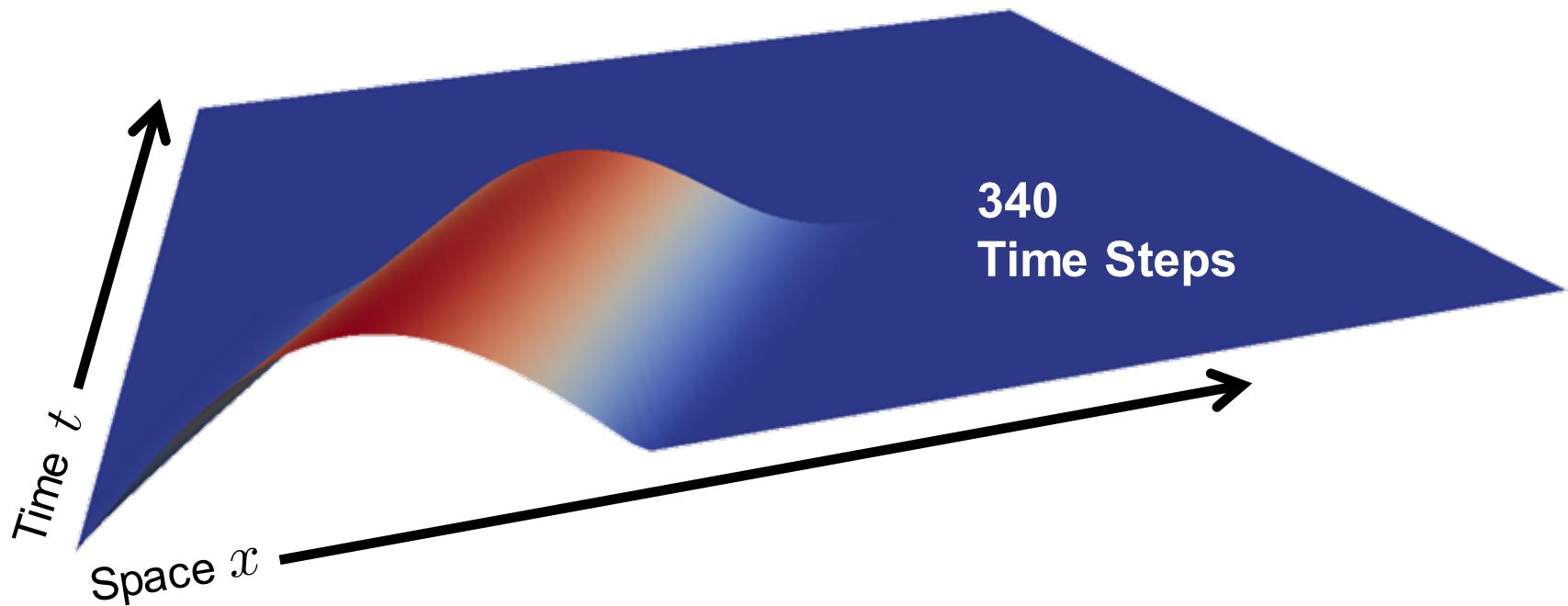
Time stepping is sequential

- Simple advection equation, $u_t = -cu_x$
- Initial condition is a wave



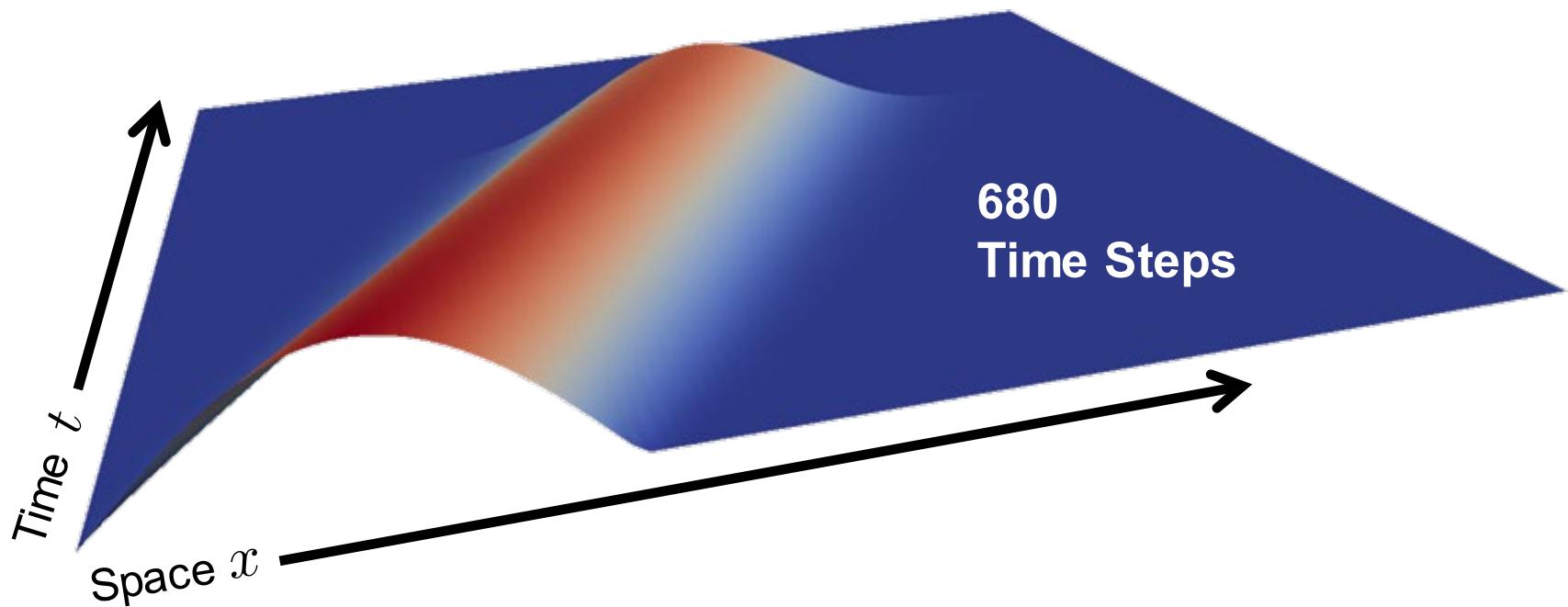
Time stepping is sequential

- Simple advection equation, $u_t = -cu_x$
- Wave propagates serially through space



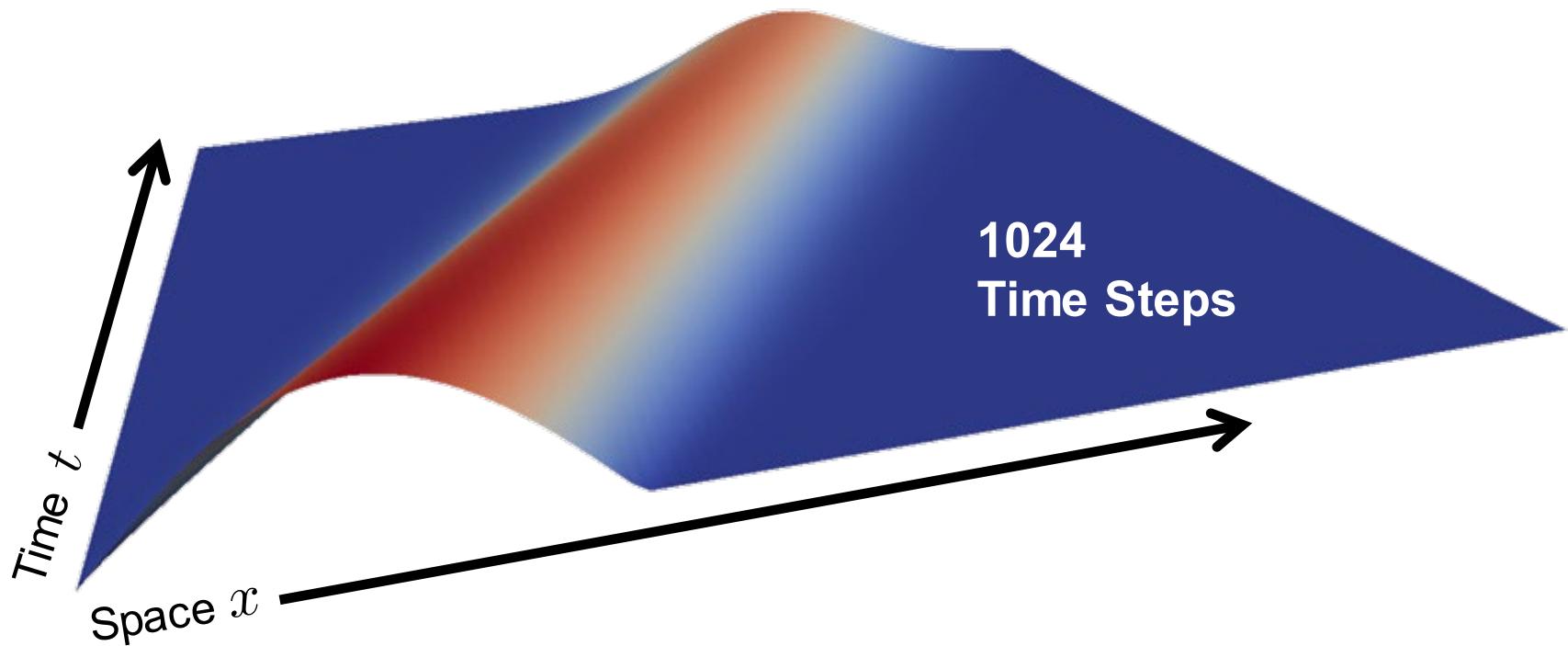
Time stepping is sequential

- Simple advection equation, $u_t = -cu_x$
- Wave propagates serially through space



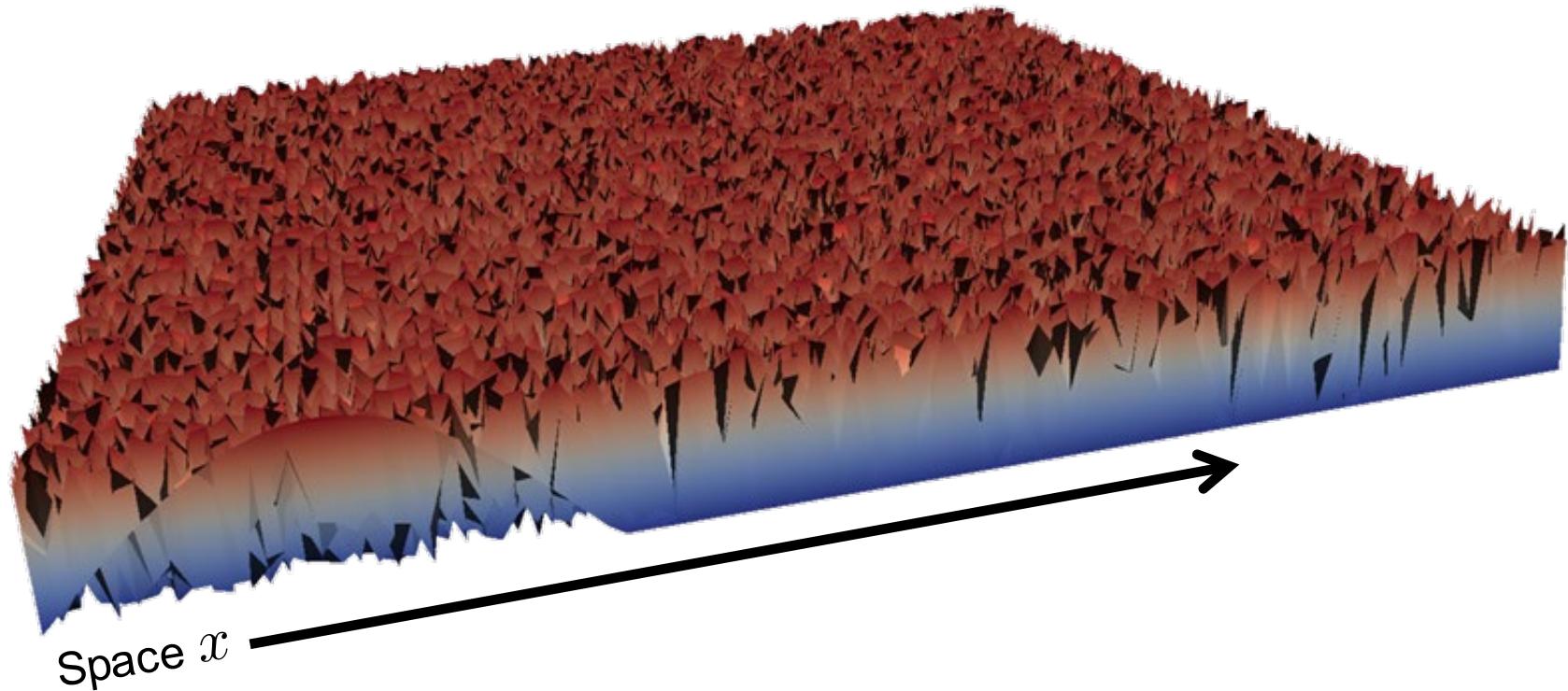
Time stepping is sequential

- Simple advection equation, $u_t = -cu_x$
- Wave propagates serially through space



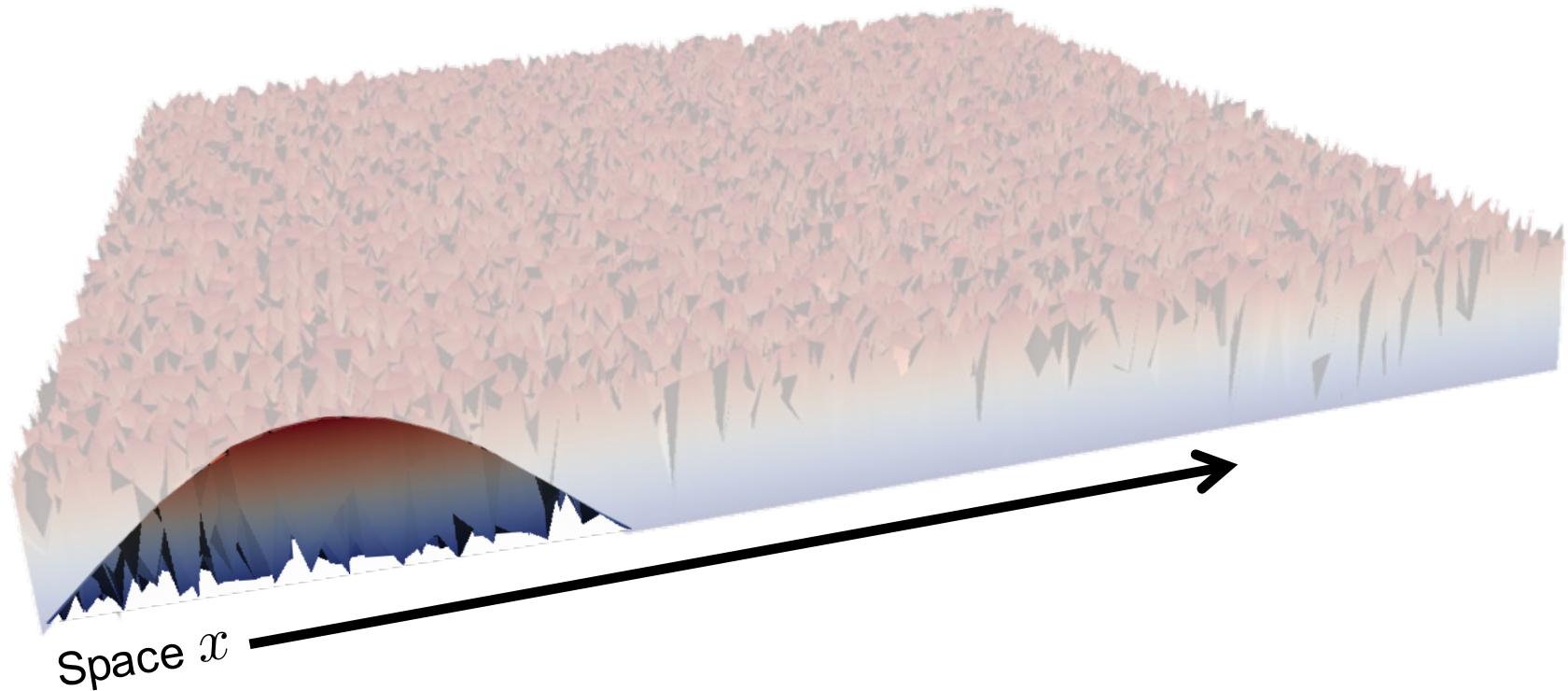
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Random initial space-time guess (only for illustration)



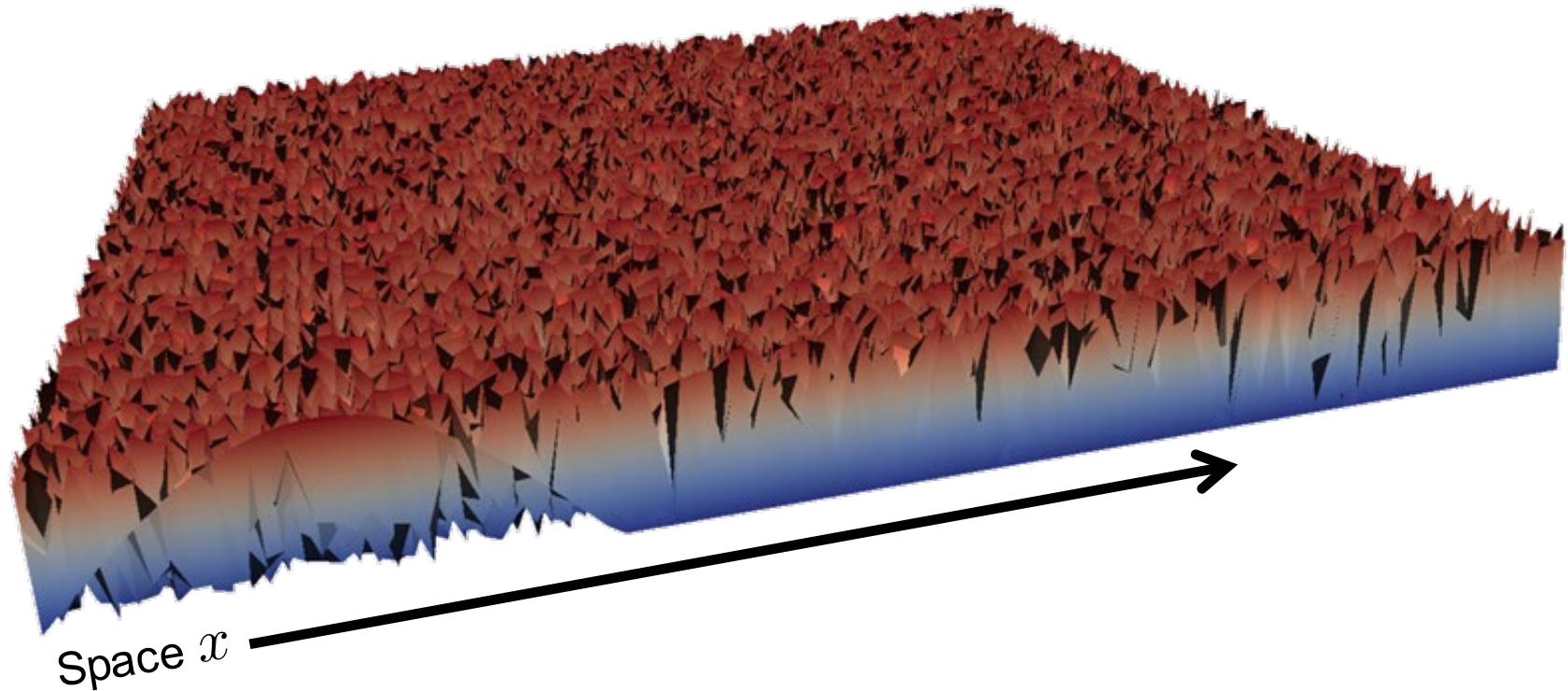
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Initial condition is a wave



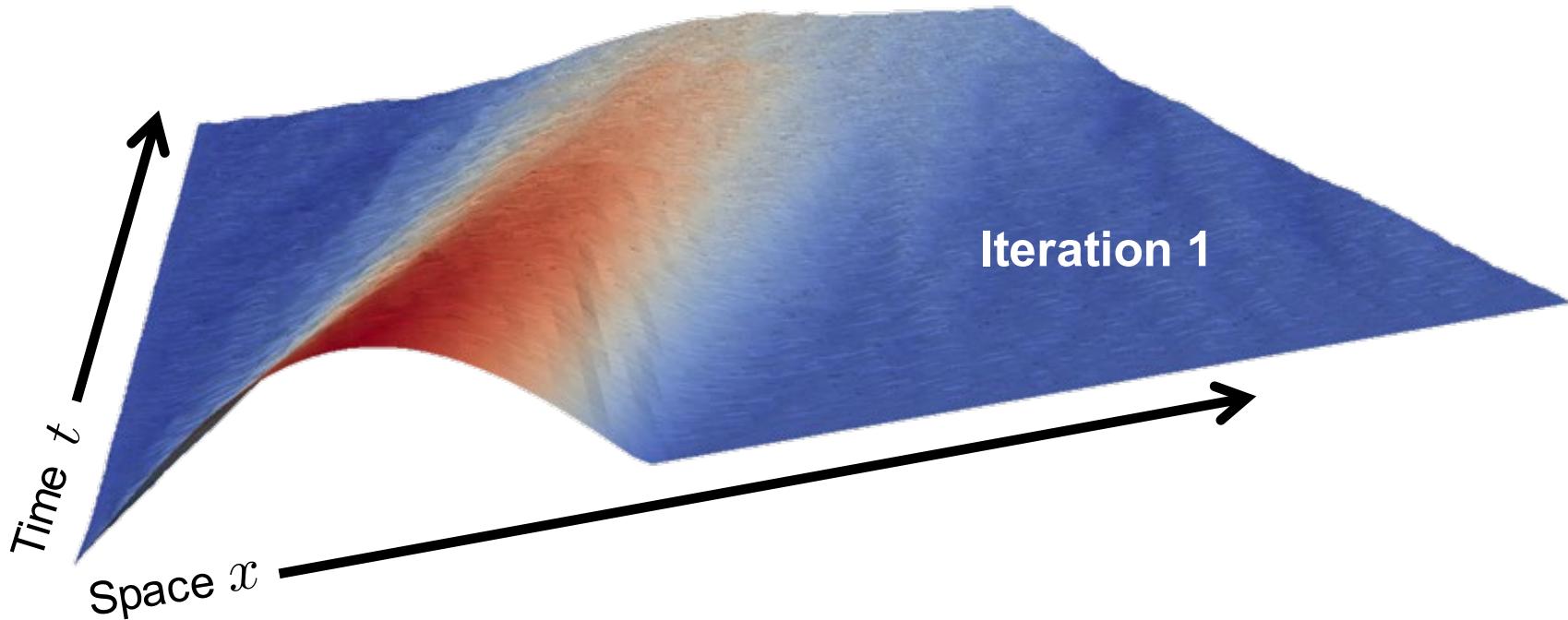
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Initial condition is a wave



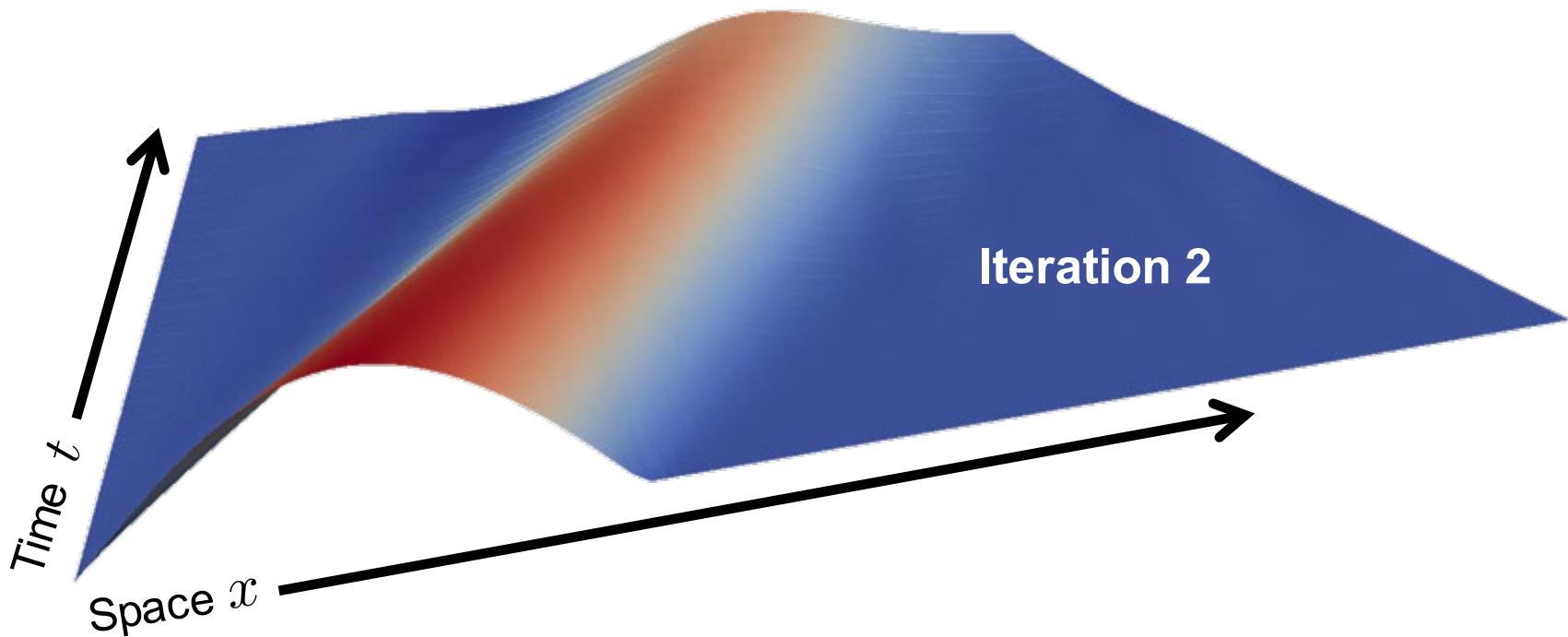
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Multilevel structure allows for fast data propagation



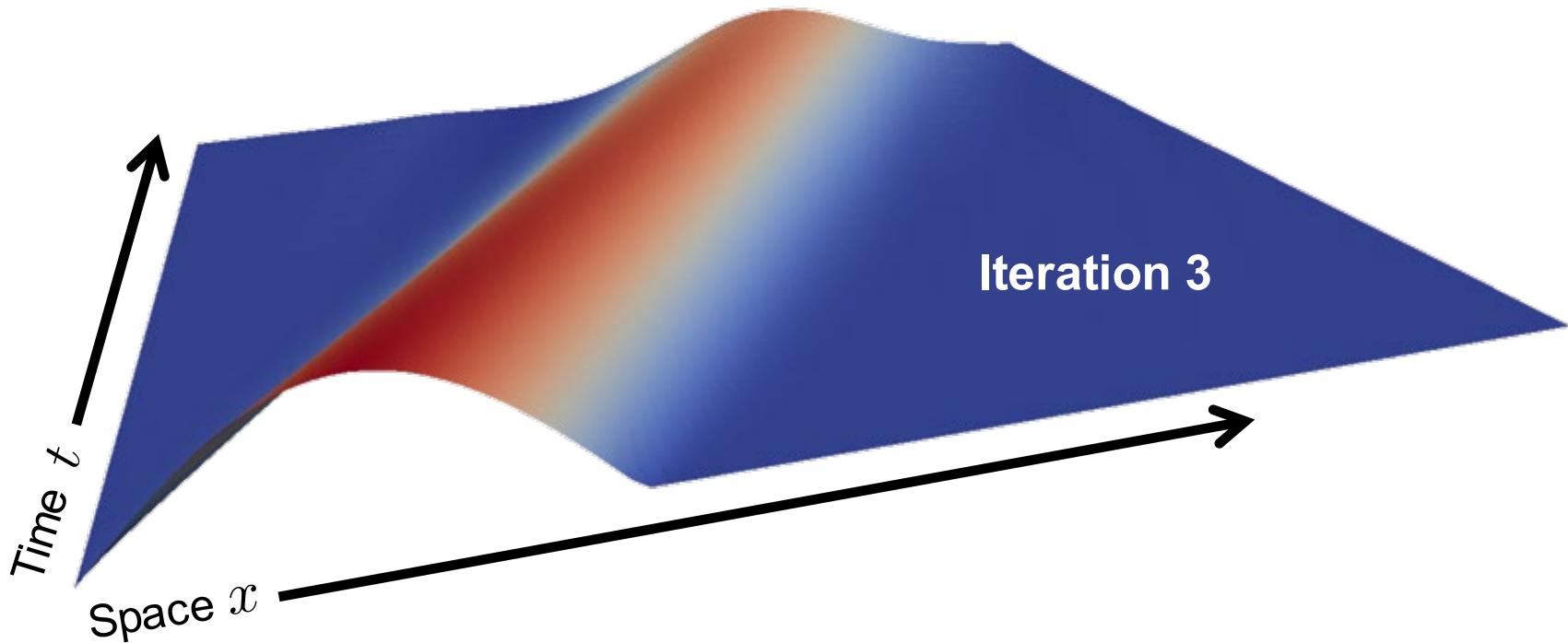
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Multilevel structure allows for fast data propagation



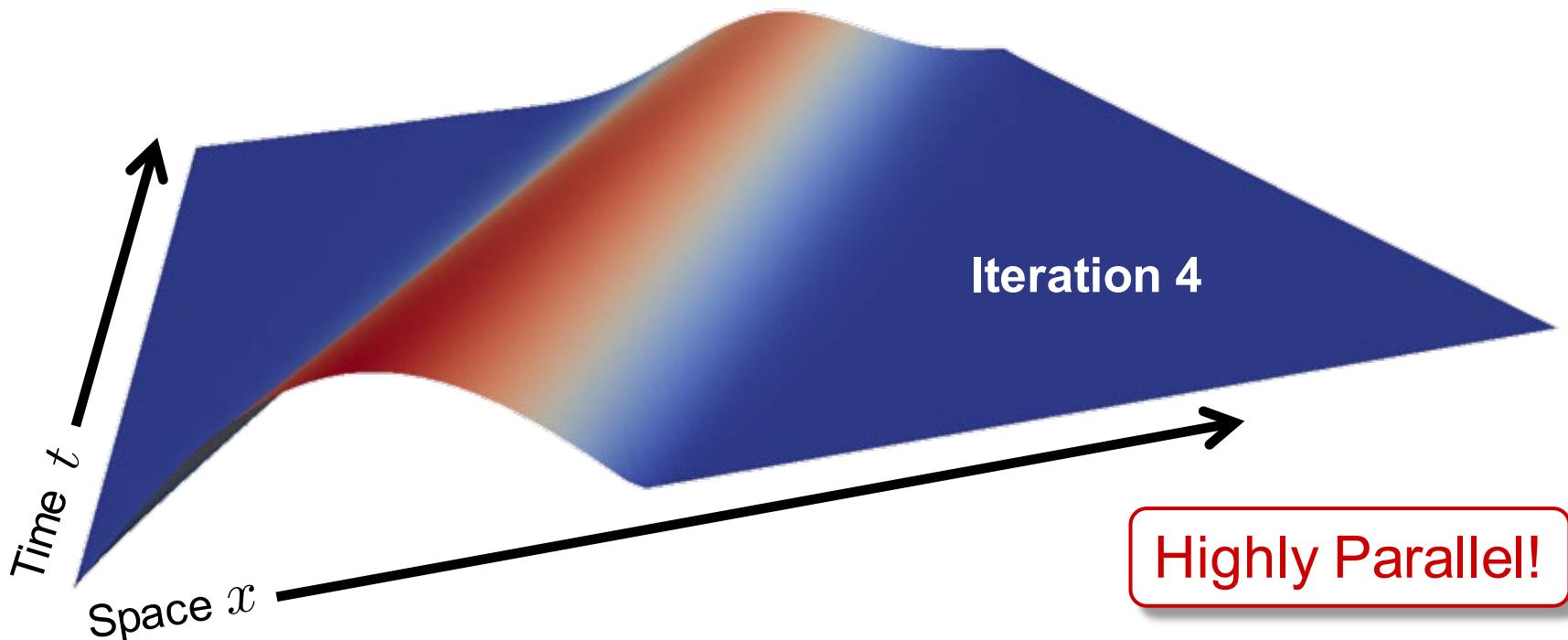
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Multilevel structure allows for fast data propagation



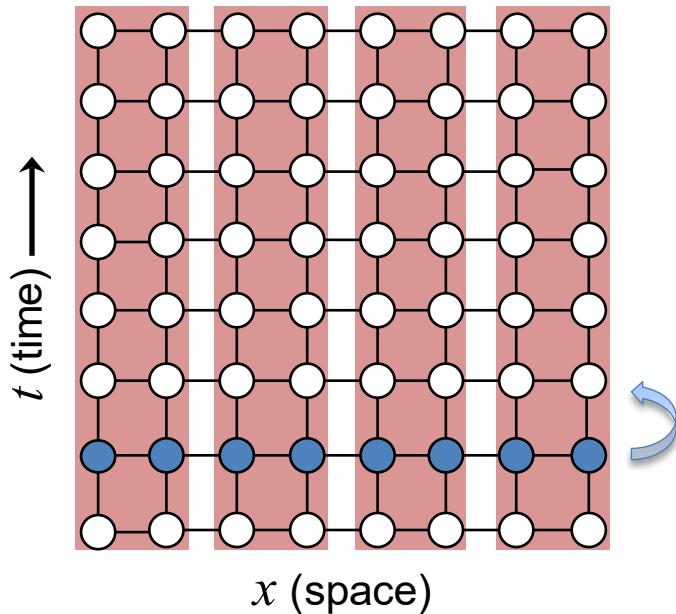
Multigrid-in-time converges to the serial space-time solution in parallel

- Simple advection equation, $u_t = -cu_x$
- Already very close to the solution

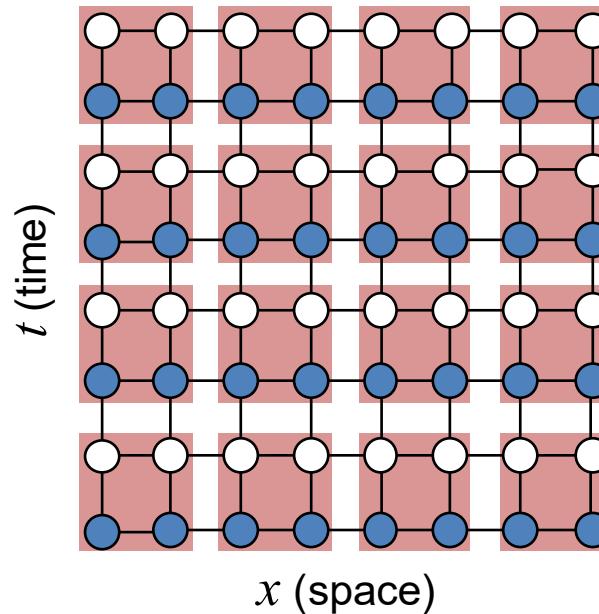


Significantly more parallel resources can be exploited with multigrid in time

Serial time stepping



Multigrid in time



⊖ Parallelize in **space only**

⊕ Store **only one time step**

⊕ Parallelize in **space and time**

⊖ Store **several time steps**

It's useful to view the time integration problem as a large block matrix system

- General one-step method

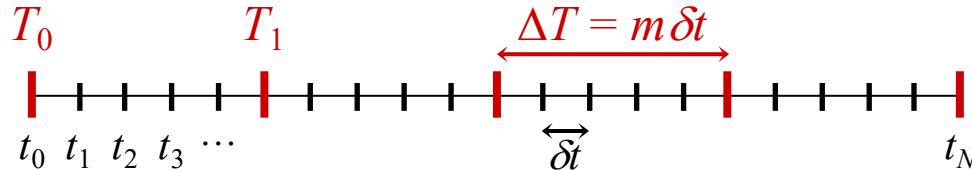
$$\mathbf{u}_i = \Phi_i(\mathbf{u}_{i-1}) + \mathbf{g}_i, \quad i = 1, 2, \dots, N$$

- Linear setting: time marching = block forward solve
 - $O(N)$ direct method, but sequential

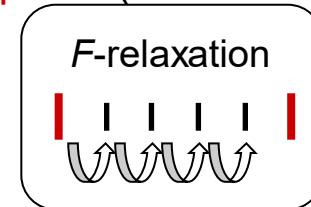
$$A\mathbf{u} \equiv \begin{pmatrix} I & & & \\ -\Phi & I & & \\ & \ddots & \ddots & \\ & & -\Phi & I \end{pmatrix} \begin{pmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_N \end{pmatrix} = \begin{pmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_N \end{pmatrix} \equiv \mathbf{g}$$

- The MGRIT approach is based on multigrid reduction (MGR) methods (approximate cyclic reduction)
 - $O(N)$ iterative method, but highly parallel
 - Non-intrusive – user only provides time integrator Φ

Parallel Multigrid in Time



- F -point (fine grid only)
- C -point (form coarse grid)



- Relaxation is highly parallel
 - Alternates between F -points and C -points
 - F -point relaxation = integration over each coarse time interval
- Define coarse-grid system with $N_\Delta = N/m$ grid points

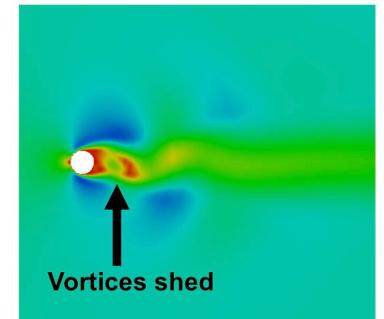
$$\mathbf{A}_\Delta(\mathbf{u}_\Delta) = \begin{pmatrix} \mathbf{I} & & & \\ -\Phi_{\Delta,1} & \mathbf{I} & & \\ & \ddots & \ddots & \\ & & -\Phi_{\Delta,N_\Delta} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{u}_{\Delta,0} \\ \mathbf{u}_{\Delta,1} \\ \vdots \\ \mathbf{u}_{\Delta,N_\Delta} \end{pmatrix} = \begin{pmatrix} \mathbf{g}_{\Delta,0} \\ \mathbf{g}_{\Delta,1} \\ \vdots \\ \mathbf{g}_{\Delta,N_\Delta} \end{pmatrix} = \mathbf{g},$$

where $\Phi_{\Delta,i}$ should be at least as cheap to apply as Φ_i

The MGRIT approach builds as much as possible on existing codes and technologies

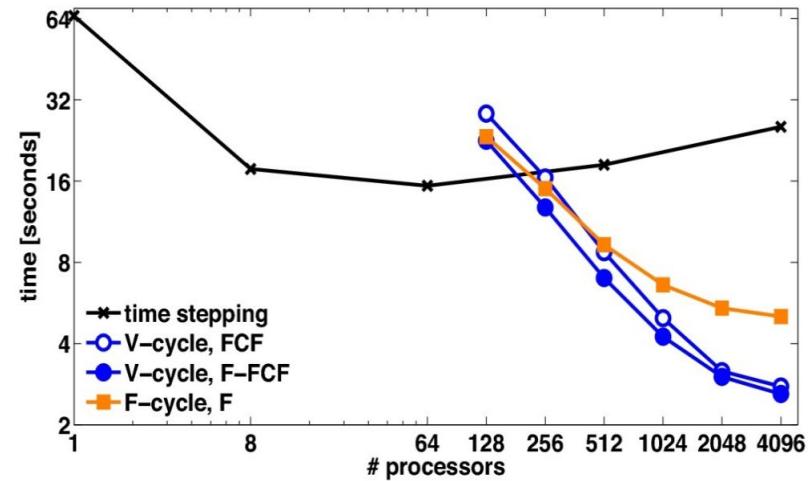
- Combines algorithm development, theory, and software proof-of-principle
- Goal: Create **concurrency** in the time dimension
- **Non-intrusive**, with unchanged time discretization
 - Implicit, explicit, multistep, multistage, ...
- Converges to **same solution** as sequential time stepping
- Extends to **nonlinear** problems with FAS formulation
- XBraid is our open-source implementation of MGRIT
 - User defines two objects and writes several wrapper routines (Step)
 - Only stores C -points to **minimize storage**
- Many active research topics, applications, and codes
 - Adaptivity in space and time, moving meshes, BDF methods, ...
 - Linear/nonlinear diffusion, advection, fluids, power grid, elasticity, ...
 - MFEM, hypre, Strand2D, Cart3D, LifeV, CHeart, GridDyn

$$\begin{pmatrix} I & & & \\ -\Phi & I & & \\ \ddots & \ddots & \ddots & \\ & -\Phi & I \end{pmatrix}$$



Parallel speedups can be significant, but in an unconventional way

- Parallel time integration is driven entirely by hardware
 - Time stepping is already $O(N)$
- Useful only beyond some scale
 - There is a **crossover point**
 - Sometimes need significantly more parallelism just to break even
 - **Achievable efficiency** is determined by the space-time **discretization** and degree of **intrusiveness**
- The **more time steps**, the **more speedup potential**
 - Applications that require lots of time steps benefit first
 - Speedups (so far) **up to 49x on 100K cores**



3D Heat Equation: $33^3 \times 4097$,
8 procs in space, **6x speedup**

Nearly 50 years of research exists but has only scratched the surface

- Earliest work goes back to **1964** by Nievergelt
 - Led to multiple shooting methods, Keller (1968)
- **Space-time multigrid** methods for parabolic problems
 - Hackbusch (1984); Horton (1992); Horton and Vandewalle (1995)
 - The latter is one of the first **optimal & fully parallelizable** methods to date
- Parareal was introduced by Lions, Maday, and Turinici in 2001
 - Probably the most widely studied method
 - Gander and Vandewalle (2007) show that parareal is a **two-level FAS multigrid** method
- Discretization specific work includes
 - Minion, Williams (2008, 2010) – PFASST, spectral deferred correction, FAS
 - DeSterck, Manteuffel, McCormick, Olson (2004, 2006) – FOSLS
- **Research on these methods continues to ramp up!**
 - Ruprecht, Krause, Speck, Emmett, Langer, ... **this is not an exhaustive list**
- Recent review: Gander (2015), “50 years of time parallel time integration”
- Lot of new research: See Session 8A

New developments for semi-structured interfaces and solvers

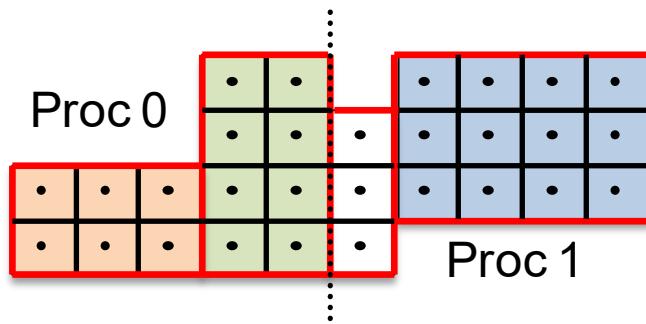


New developments for semi-structured interfaces and solvers

- How can we increase structure?
 - Develop new structured-grid matrix class that supports rectangular matrices and constant coefficients (completed, not released yet)
 - Develop new semi-structured-grid matrix class that builds on the new structured-grid matrix (in progress)
 - Develop new semi-structured algebraic multigrid solver based on PFMG
- Benefits:
 - New matrix class
 - Allows replacement of structured components in SStruct interface that are currently treated in an unstructured fashion
 - Allows development of new semi-structured algebraic multigrid solvers (interpolation, RAP); currently only AMG solver available in SStruct interface is unstructured AMG solver BoomerAMG
 - Improved performance

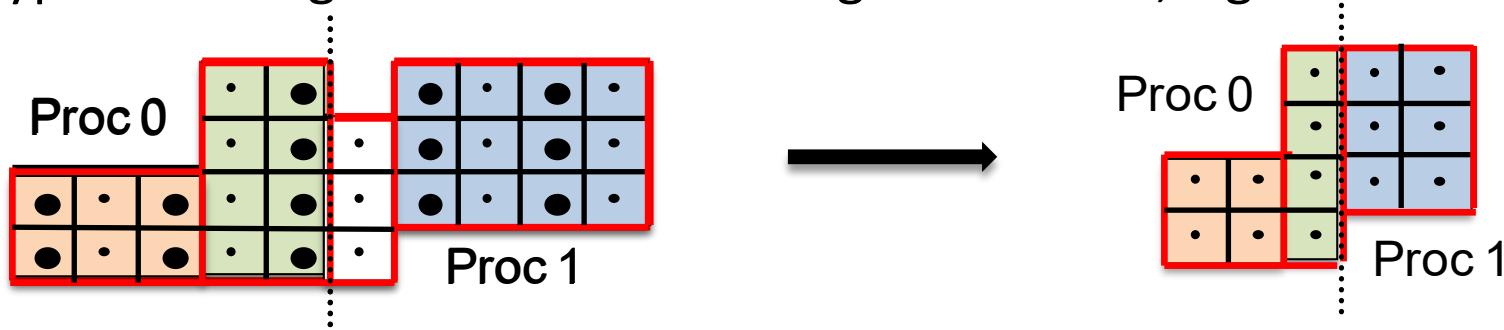
What needs to change for rectangular matrices?

- Requires two grids – domain and range grid
- Typical rectangular matrices in multigrid methods, e.g. restriction,



What needs to change for rectangular matrices?

- Requires two grids – domain and range grid
- Typical rectangular matrices in multigrid methods, e.g. restriction, e.g.

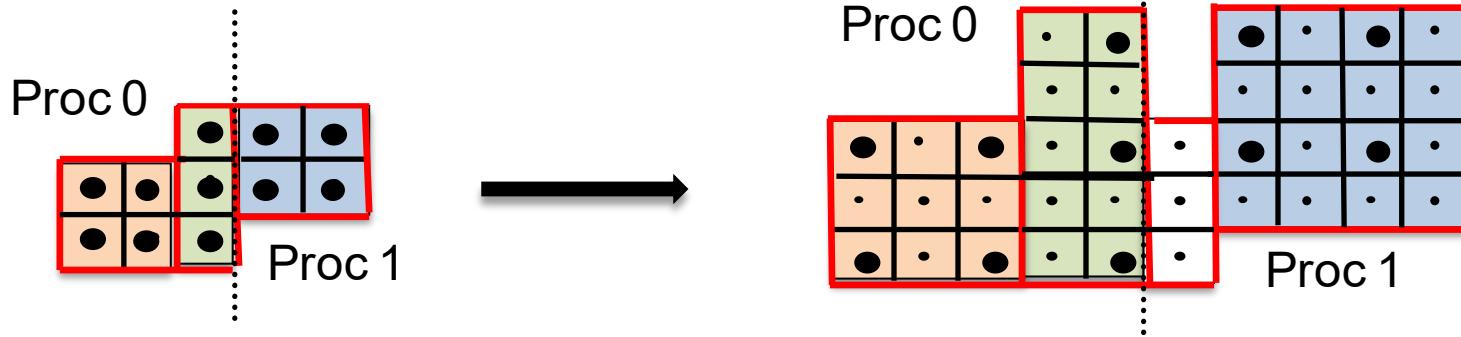


- Note that boxes as well as neighbor processes can disappear when coarsening, this requires careful alignment
- Stencil, e.g.

$$R = \begin{bmatrix} 1/2 & 1 & 1/2 \end{bmatrix}$$

How about interpolation?

- Now domain grid coarser than range



- What about stencil?

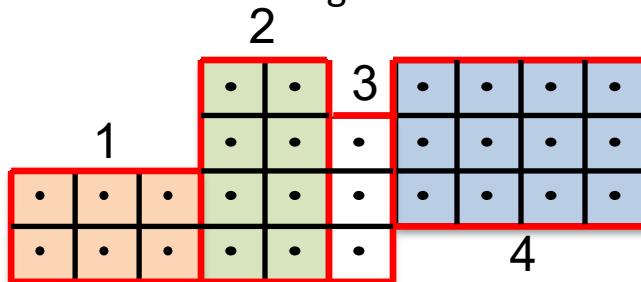
Example:

$$\mathbf{P} = \begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} 1/4 & * & 1/4 \\ * & * & * \\ 1/4 & * & 1/4 \end{bmatrix}^{r_1} \oplus \begin{bmatrix} 1/2 & * & 1/2 \end{bmatrix}^{r_2} \oplus \begin{bmatrix} 1/2 \\ * \\ 1/2 \end{bmatrix}^{r_3} \oplus [1]^{r_4}$$

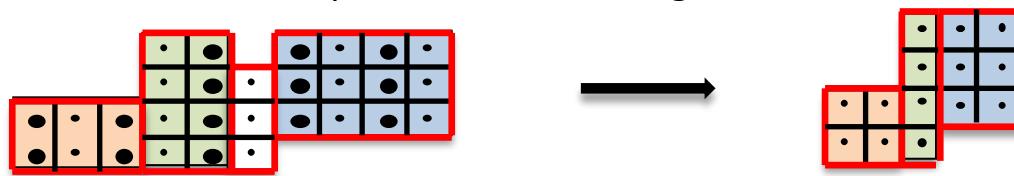
New StructMatrix Data structure

- New StructMatrix has
 - a base grid from which both grids can be derived



Note that box numbers are local to process, (proc, box-num) unique across global grid

- two strides that define potential coarsening factors:



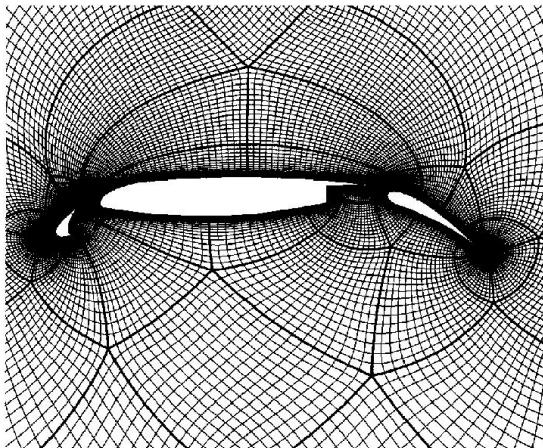
here (1,1) for domain

and (2,1) for range

- two sets of box numbers that define the subset of the base grid for domain and range
 - here (1,2,3,4) for domain
 - (for efficiency)
 - and (1,2,4) for range

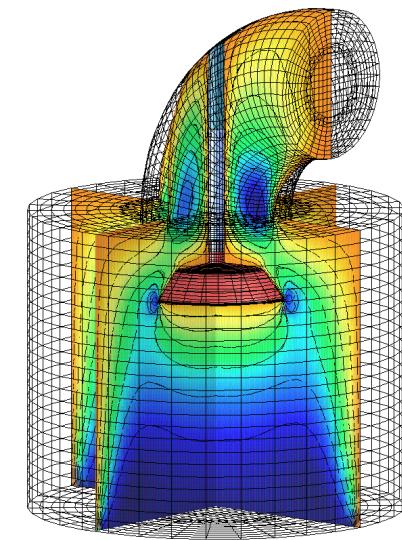
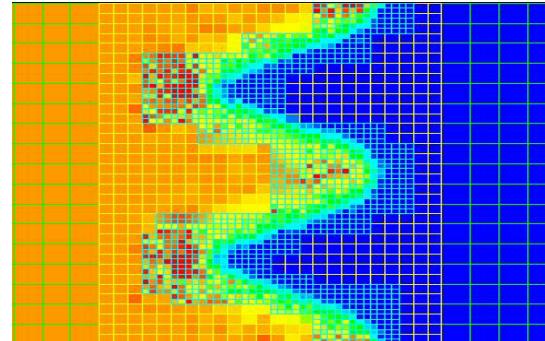
Semi-Structured-Grid System Interface (SStruct)

- Allows more general grids:
 - Grids that are mostly (but not entirely) structured
 - Examples: *block-structured grids, structured adaptive mesh refinement grids, overset grids*



Block-Structured

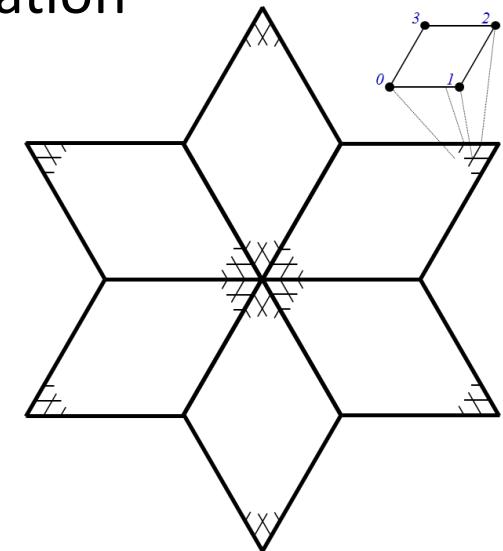
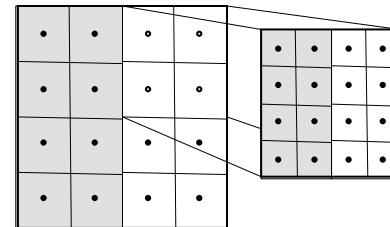
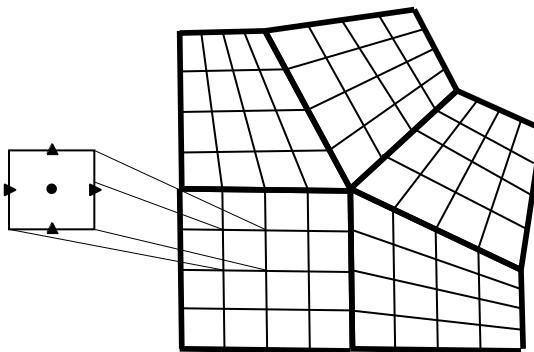
*Adaptive Mesh
Refinement*



Overset

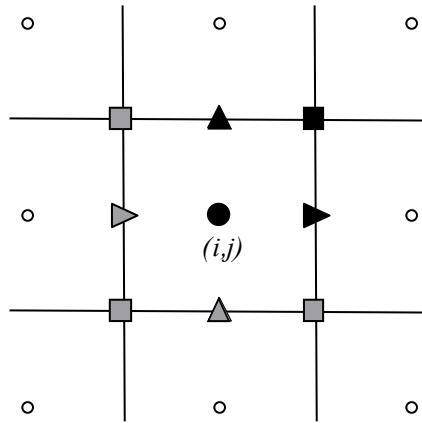
Semi-Structured-Grid System Interface (SStruct)

- The SStruct grid is composed out of structured grid *parts*
- The interface uses a *graph* to allow nearly arbitrary relationships between part data
- The graph is constructed from stencils or finite element stiffness matrices plus additional data-coupling information

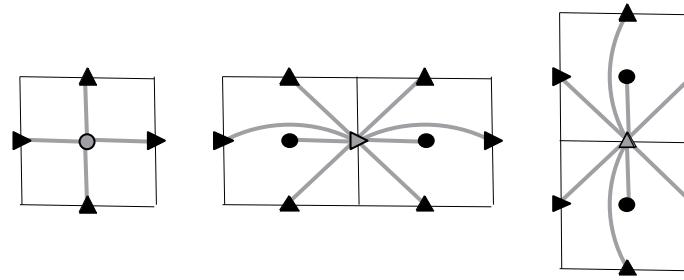


Semi-Structured-Grid System Interface (SStruct)

- Allows more general PDE's
 - Multiple variables (system PDE's)
 - Multiple variable types (cell centered, face centered, vertex centered, ...)



3 discretization stencils



- Relationships between different variable types require rectangular matrices and are currently treated as unstructured, can now be replaced with structured matrices

SStructMatrix data structure

- The SStructMatrix data structure is based on a splitting of the nonzeros into structured and unstructured couplings:

$$A = S + U$$

- S is stored as a collection of struct matrices for each part and variable type and currently contains only stencil couplings between the same variable type

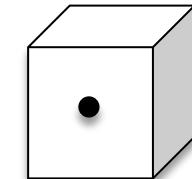
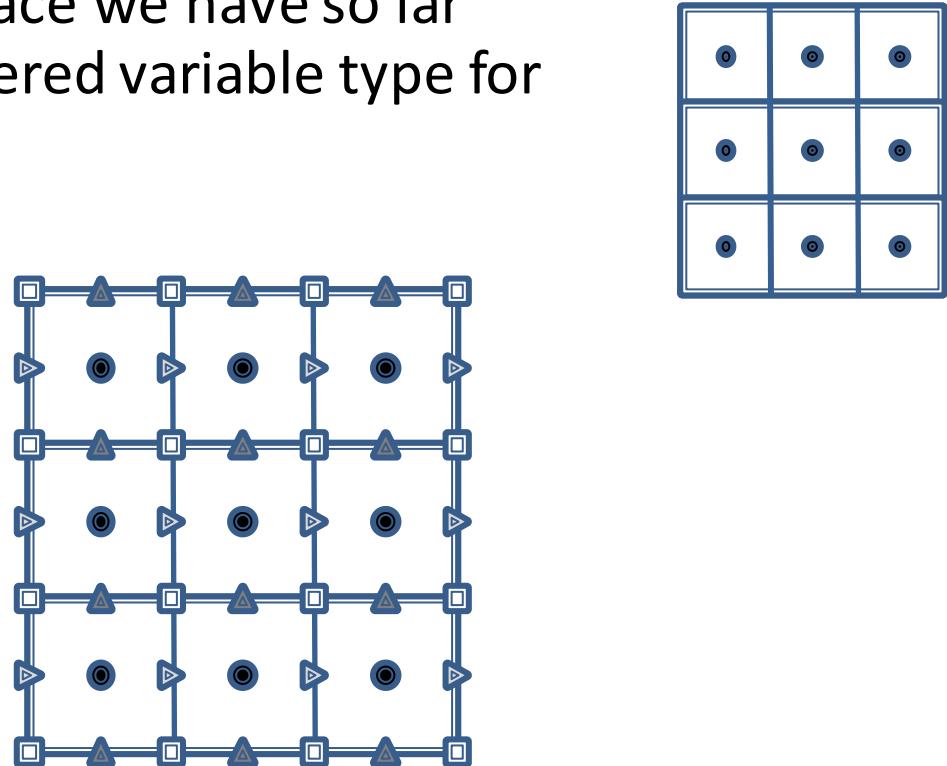
submatrix for a
part with
3 variable types

$$\begin{pmatrix} S_{11} & U_{12} & U_{13} \\ U_{21} & S_{22} & U_{23} \\ U_{31} & U_{32} & S_{33} \end{pmatrix} \rightarrow \begin{pmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{pmatrix}$$

- Semi-structured rectangular matvec comes with additional difficulties!

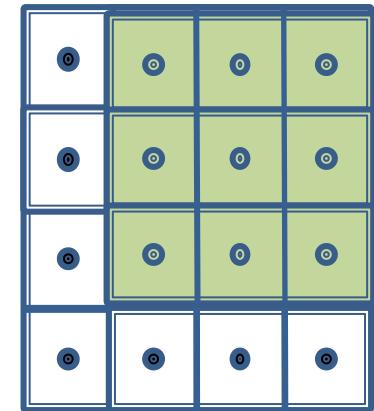
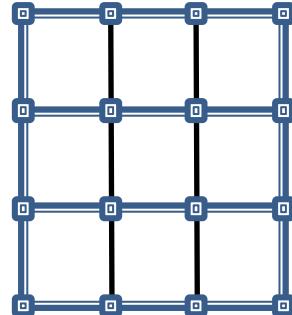
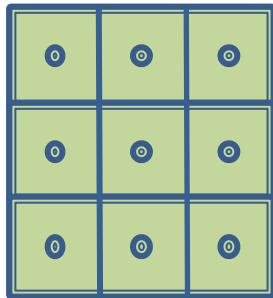
Boxes for different variable types

- In context of Struct interface we have so far considered only cell-centered variable type for matvec
- Now we also have face- and edge-centered
- node-centered
- and potentially more in higher dimensions !



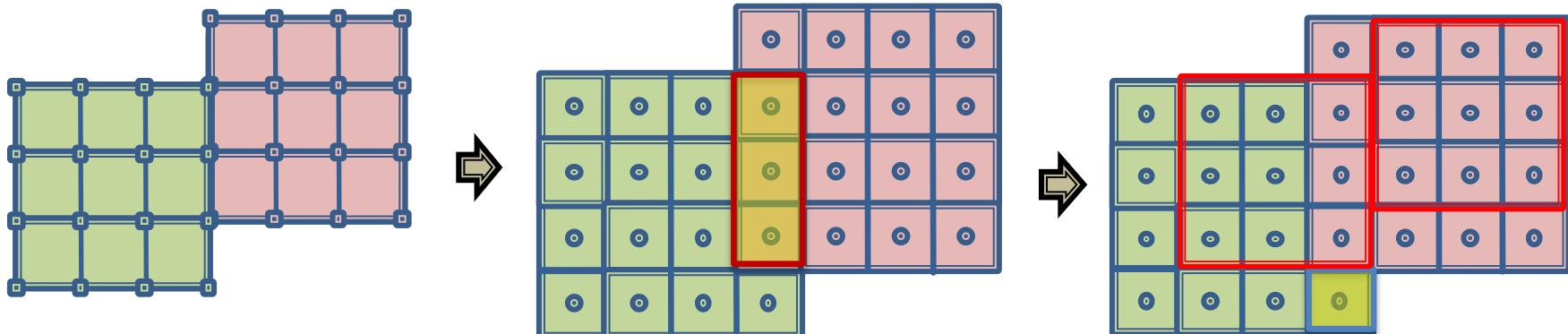
Rectangular matrix between different variable types

- Need to somehow be able to consider them in the context of boxes of struct interface
- Difficulty in properly lining up boxes for different variable types
- How can we match them up properly?
- Current approach:

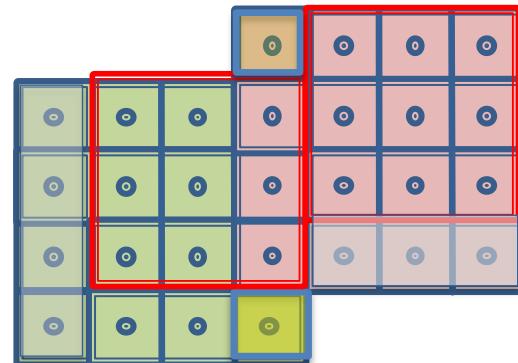


What about a group of boxes?

- Consider nodal variable type for this grid

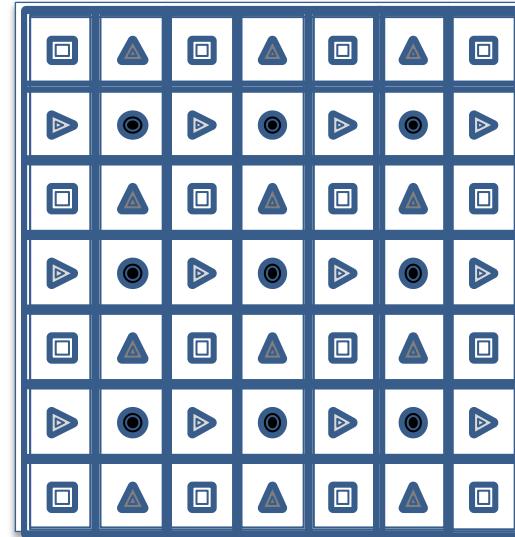


- But not allowed to overlap!
- Current algorithm to deal with this within the same variable type
- Becomes too complicated for a combination of variable types



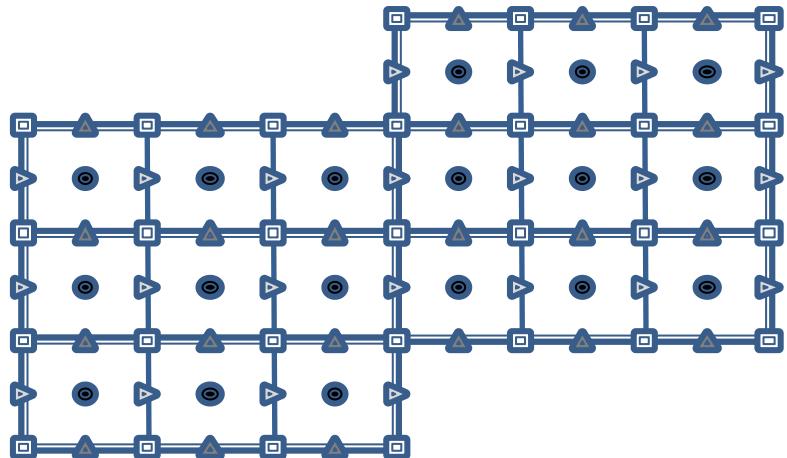
Better way to deal with this:

- See Engwer, Falgout, Yang, *Concurrency and Computation: Practice and Experience* 2017
- Better option:
Refine the grid and use strides!
Now one can use the old
method!



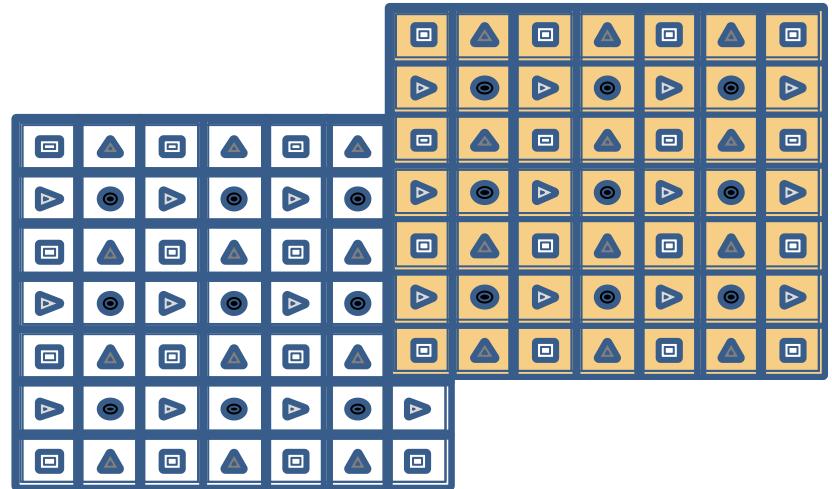
Better way to deal with this:

- See Engwer, Falgout, Yang, *Concurrency and Computation: Practice and Experience* 2017
- Better option:
Refine the grid and use strides!
Now one can use the old
method!



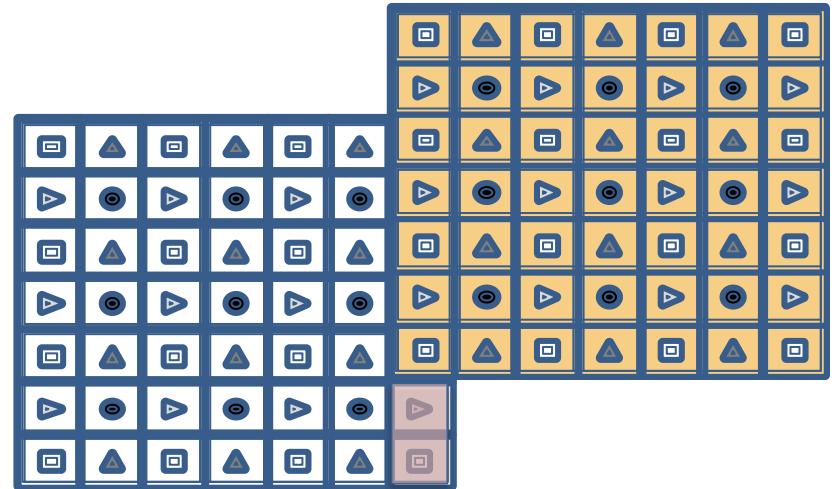
Better way to deal with this:

- See Engwer, Falgout, Yang, *Concurrency and Computation: Practice and Experience* 2017
- Better option:
Refine the grid and use strides!
Now one can use the old
method!



Better way to deal with this:

- See Engwer, Falgout, Yang, *Concurrency and Computation: Practice and Experience* 2017
- Better option:
Refine the grid and use strides!
Now one can use the old
method!
- We still need to work out
the details on this!



Semi-Structured Algebraic Multigrid (SSAMG)

- Straightforward to implement with completion of new SStruct interface
- Will be based on PFMG



Semi-Structured Algebraic Multigrid Solver

- SStructMatrix is split into structured and unstructured couplings:
 $A = S + U$, where U is a very small portion of the matrix
- Generate interpolation and restriction for structured parts only:

$$P_S, \quad R_S = P_S^T$$

Adjust weights at part boundaries as needed

- Generate coarse grid operator:

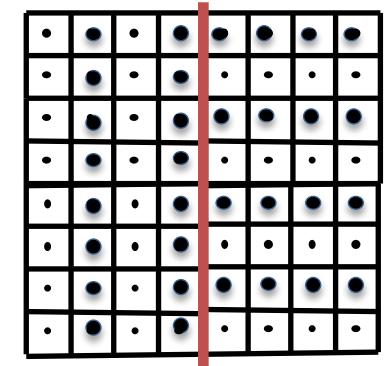
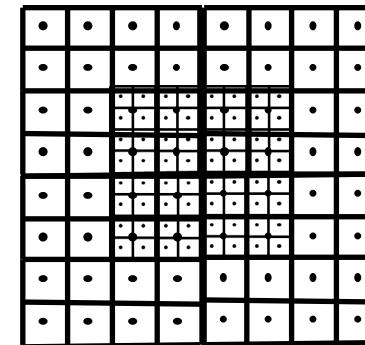
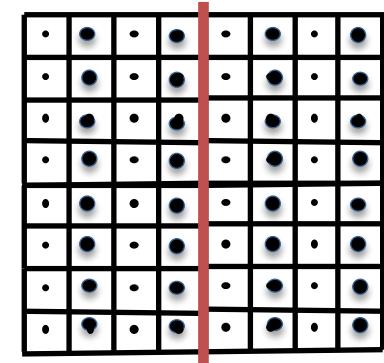
$$A_C = R_S A P_S = R_S S P_S + R_S U P_S = S_C + U_C$$

mostly structured ops with good potential for efficient performance

- With 2-point interpolation, U_C continues to be restricted to part boundaries, no growth into interior
- Apply weighted Jacobi smoother to $S_C + U_C$

Experiments and Test Problems

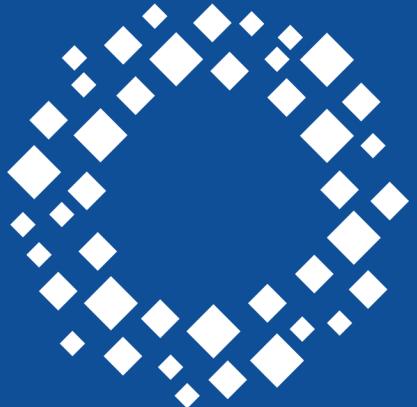
- Tested this method for $u_{xx} + cu_{yy} (+u_{zz}) = b$ as preconditioner for CG
- Weighted Jacobi smoother, $w=0.67$
- Set c to 1 and 0.01
- Also tried a mixed anisotropy problem, anisotropies 0.001
- AMR problem (coarsen to grid length $n=4$)
- Compared to
 - PFMG-CG on whole domain
 - AMG-CG
(Classical RS, P trunc. to 2 nnz/row
→ similar complexities as SSAMG)



Convergence results for grid sizes $n \times n$ in 2D and $n \times n \times n$ in 3D

	PFMG			AMG			SSAMG		
n	32	64	128	32	64	128	32	64	128
C=1; 2D	8	8	9	11	12	14	9	12	15
C=0.01; 2D	6	6	7	6	7	8	13	15	18
Mix; 2D	55	98	145	11	13	16	16	22	31
AMR; 2D	-	-	-	11	13	15	13	16	18
C=1; 3D	9	9	9	12	17	26	9	10	11
C=0.01; 3D	8	8	8	8	9	11	9	11	13
Mix; 3D	36	68	107	11	13	17	12	14	20
AMR; 3D	-	-	-	13	17	22	16	18	20

- Achieves better scalability in 3D than 2D
- Future research will investigate inclusion of unstructured portion into interpolation: $P_S + P_U$



CASC

Center for Applied
Scientific Computing



**Lawrence Livermore
National Laboratory**

Thank You!

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.