High Performance Computing Workshop

Session 4: Algorithms and Scalability

David Keyes
KAUST and Columbia University

Philosophy of presentation

- Applications are *given* (evolving over time; see John's lectures)
- Architectures are *given* (evolving over time; see Bill's lectures)
- Algorithms *must be adapted or created* to bridge to the hostile architectures to execute the complex applications
  - as important *as ever* today, with transformation of Moore's Law from speed-based to concurrency-based
  - algorithmic concurrency main driver, but programming model stresses arise, too, when on-chip memories are shared in "multicore" architectures
- Algorithms packaged in reliable, portable software allow non-experts to compute like experts (see my second lecture)
- Knowledge of software availability and their performance vulnerability factors can usefully feed back to influence:
  - the way applications are formulated
  - the way architectures are constructed


The words of a sage

"We shape our buildings; thereafter they shape us."

Winston Churchill, 1943

our "buildings" = our computers

our computers → already arrived at the petascale in at least four architectural designs, but not yet routinely used by scientists and engineers

Progression of high performance computing

Capability

- For computation involving systems with multiple scales
- Spatial scale determines required work per "computation step" (cells, particles, agents, etc.)
- Temporal scale determines number of steps

Complexity

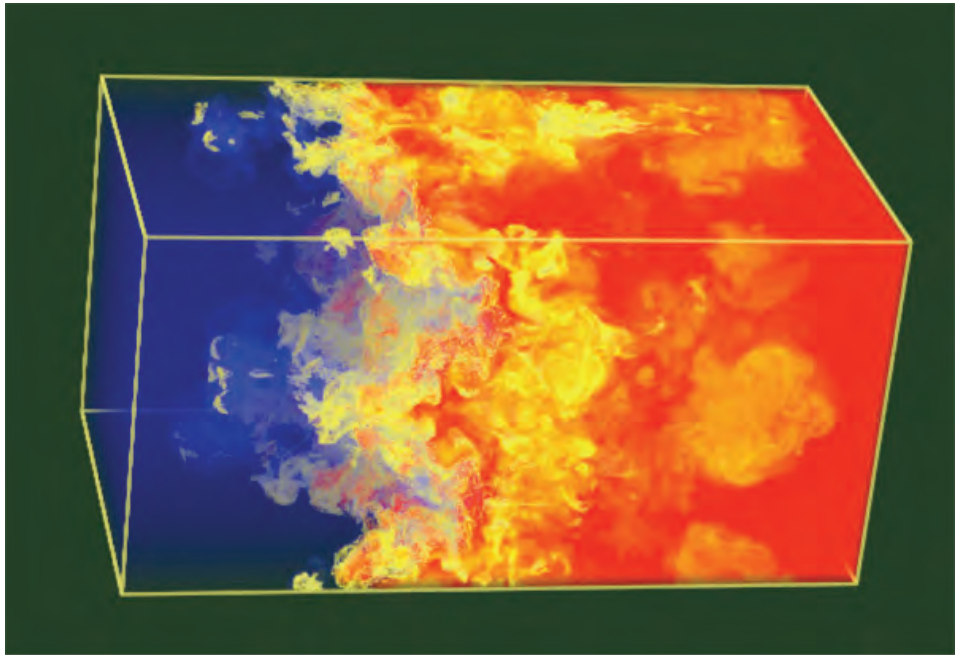
- For computation combining multiple components to produce an integrated model of a complex system
- Individual components may each require high capability
- Coupling between components requires that all components be modeled simultaneously

Understanding

- For computations executed many times with varying model parameters, inputs and boundary conditions
- Goal is to develop a clear understanding of behavior / dependencies / sensitivities of the solution over a range of parameters, which may be uncertain

Capability – multiple scales

- **Multiple spatial scales**
  - **interfaces, fronts, layers**
  - **thin relative to domain size, $\delta << L$**
- **Multiple temporal scales**
  - **fast waves**
  - **small transit times relative to convection or diffusion, $\tau << T$**

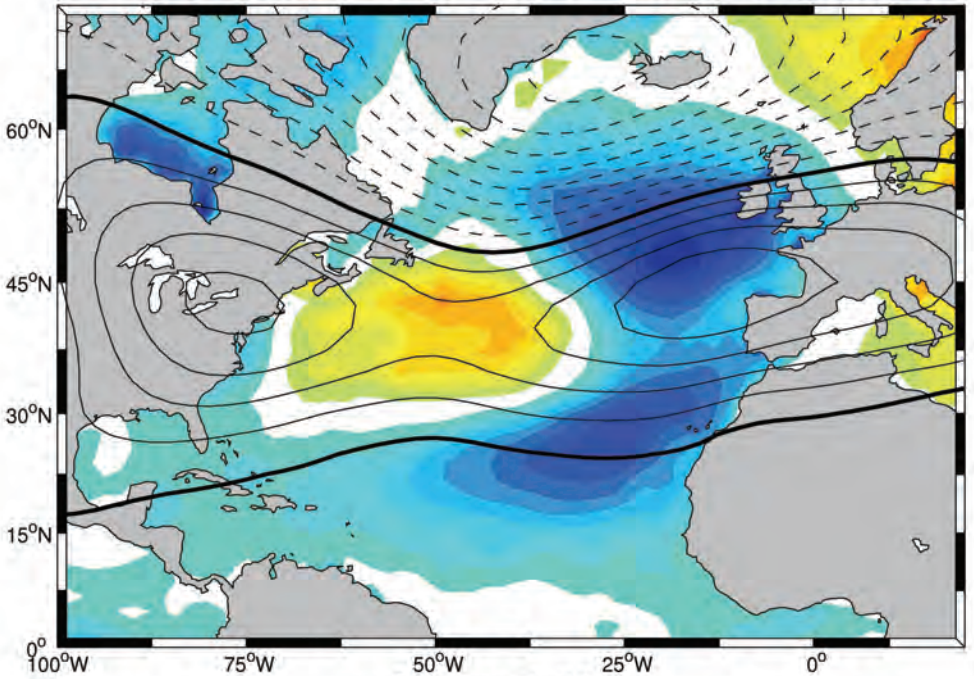


Richtmeyer-Meshkov instability, c/o A. Mirin, LLNL

- **Analyst must first *isolate dynamics of interest* and *model the rest* in a system that can be discretized over more modest range of scales**
- **Often involves filtering of high frequency modes, quasi-equilibrium assumptions, etc.**
- **May lead to infinitely "stiff" subsystem requiring implicit treatment**
- **Resulting implicit subsystem may be very ill-conditioned**

Complexity – multiple components

- **Interfacial coupling examples**
  - **Ocean-atmosphere coupling in climate**
  - **Core-edge coupling in tokamaks**
  - **Fluid-structure vibrations in aerodynamics**
  - **Boundary layer-bulk phenomena in fluids**
  - **Surface-bulk phenomena in solids**



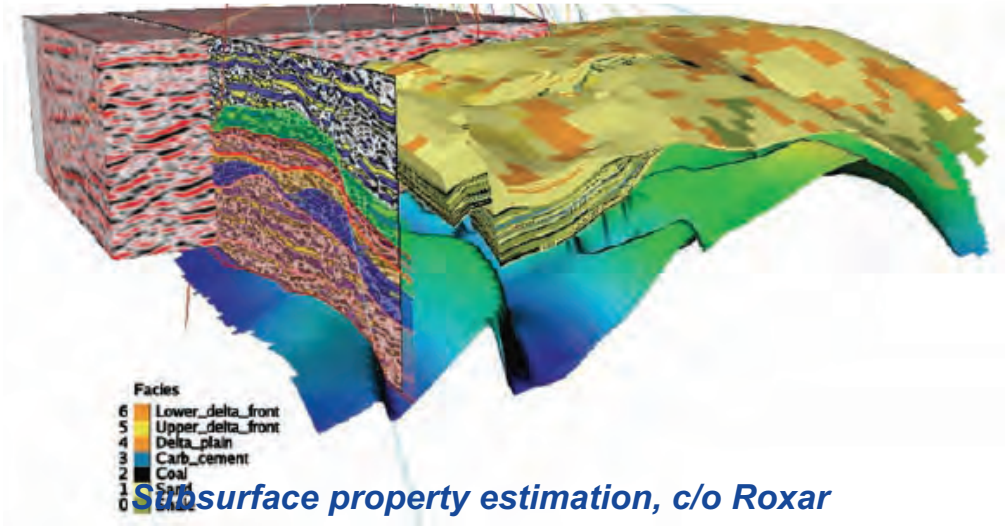
SST Anomalies, c/o A. Czaja, MIT

- **Bulk-bulk coupling examples**
  - **Radiation-hydrodynamics**
  - **Magneto-hydrodynamics**

- **Coupled systems may admit destabilizing modes not present in either system alone**

Understanding – multiple parameter sets

- Subsurface contaminant transport and petroleum recovery
- Climate prediction
- Medical imaging
- Stellar dynamics, e.g., supernovae
- Waves in inhomogeneous media
- Nondestructive evaluation of bridges, other structures



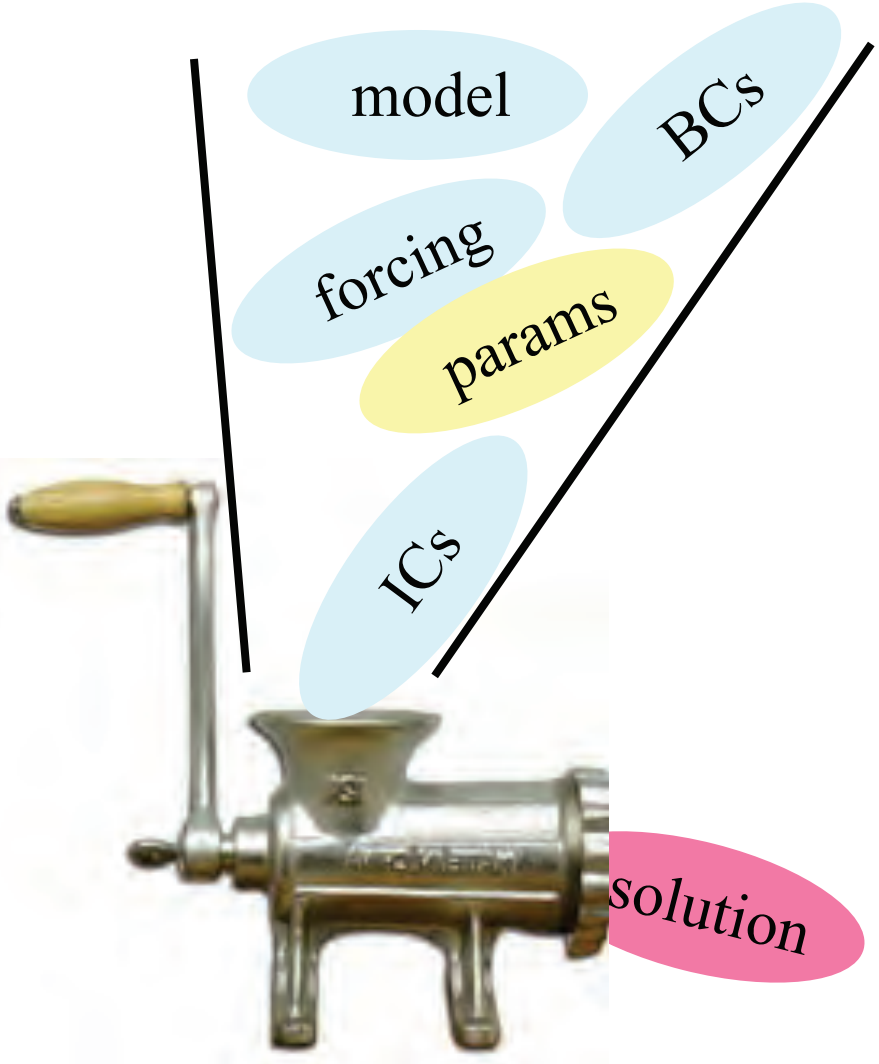
Subsurface property estimation, c/o Roxar

- Uncertainty can be in
  - constitutive laws
  - initial conditions
  - boundary conditions

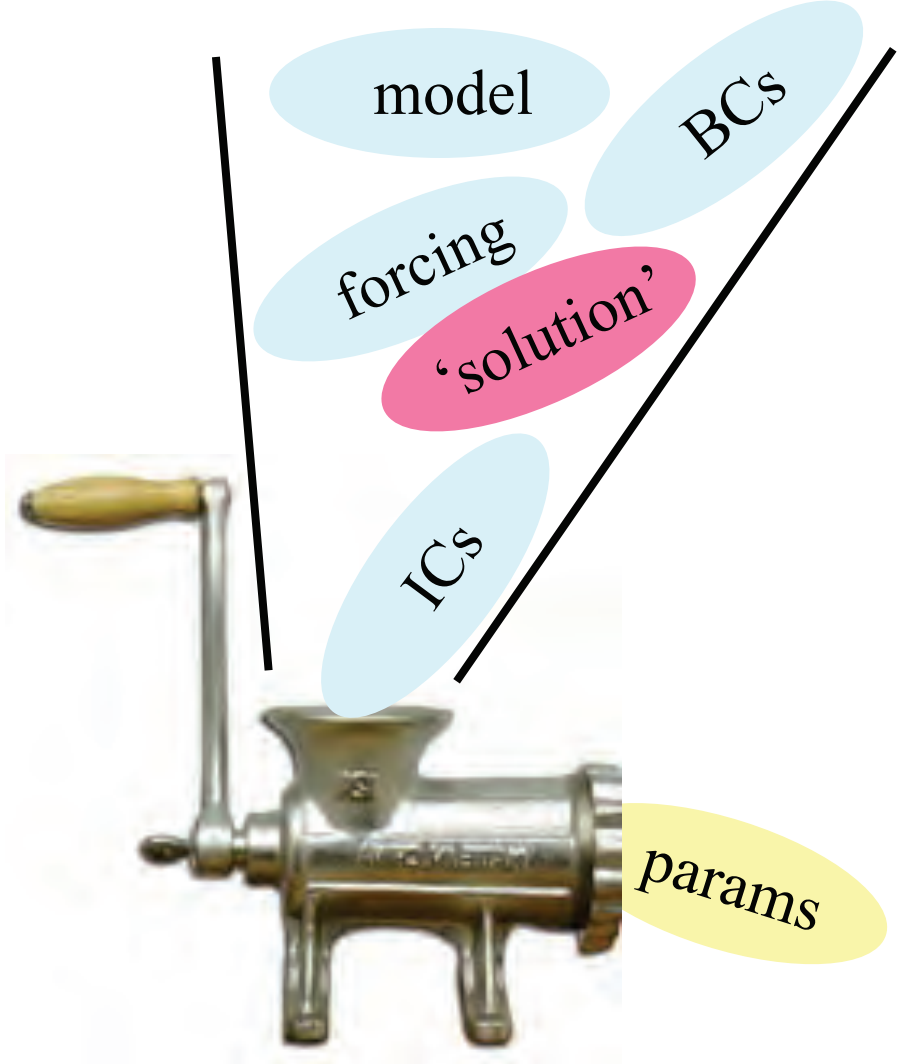
- Sensitivity, optimization, parameter estimation, boundary control all require the ability to apply the inverse action of the Jacobian of the system – a capability present in all Newton-like implicit methods

Forward vs. inverse problems

forward problem



inverse problem



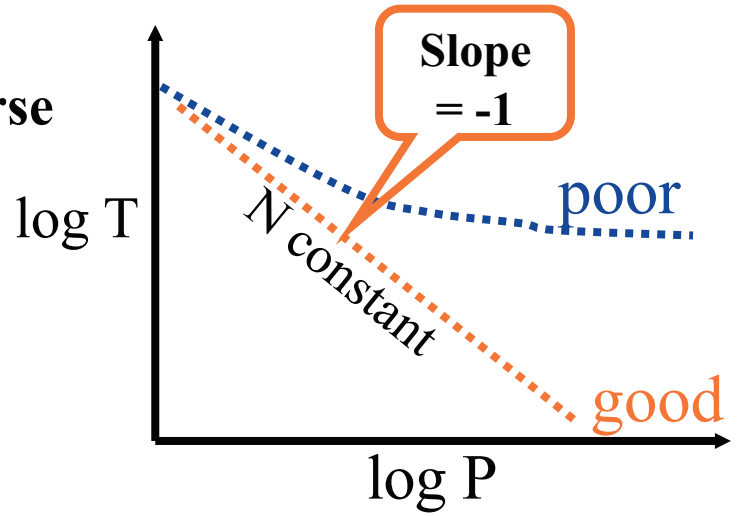
+ regularization

Top reasons simulations must scale

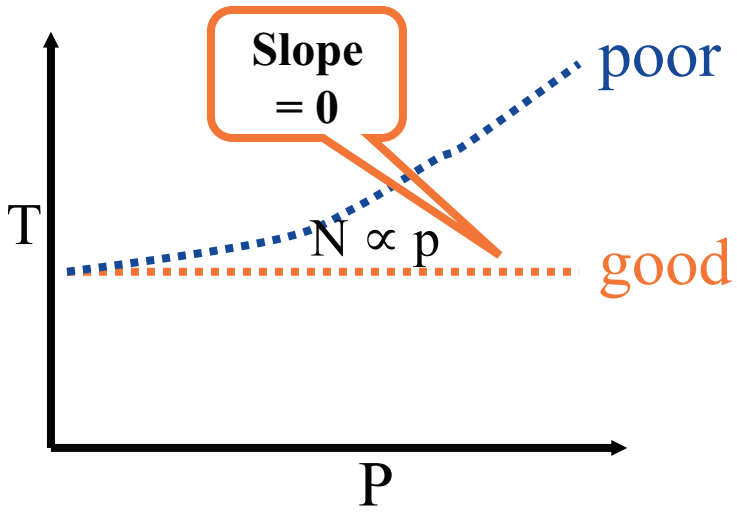
- Better resolve the full, natural range of length or time scales in a model (even if only to tune a reduced model)
- Accommodate physical effects with greater fidelity
- Allow the model degrees of freedom in all relevant dimensions
- Better isolate artificial boundary conditions or better approach realistic levels of dilution
- Combine multiple complex models
- Solve an inverse problem
- Perform data assimilation
- Perform optimization or control
- Quantify uncertainty
- Improve statistical estimates
- Operate *without models*, directly, using large data sets (BlueGene as a "big fast disk")

Two colloquial definitions of scalability

- **"Strong scaling"**
  - **execution time (*T*) decreases in inverse proportion to the number of processor-memory units (*P*)**
  - ***fixed size problem* (*N*) *overall***
  - **often instead graphed as reciprocal, "speedup"**



- **"Weak scaling"**
  - **execution time remains constant, as problem size and processor number are increased in proportion**
  - ***fixed size problem per processor***
  - **also known as "Gustafson scaling" (Moler is happy to be credited, too!)**



Formal concepts of scalability

- **"Problem-constrained"**
  - **Synonym for strong scaling; emphasizes inability to fill up the machine**
- **"Memory-constrained"**
  - **A form of weak scaling in which memory per processor is held constant**
  - **Discrete size of problem grows to match processor growth, e.g., by refinement**
- **"Time-constrained"**
  - **A form of weak scaling in which execution time per processor is held constant**
  - **Differs from memory-constrained if work is not proportional to storage**
  - **Commonly, work is superlinear in storage; time-constrained scaling is then unable to fill up the machine**

Formal concepts of scalability, cont.

- **"Efficiency" is speed-up ratio between two scales divided by the ratio of processor-memory units**
  - **Efficiency less than unity reflects the inevitability that the share of time spent coordinating the computation (communicating data required on multiple processors and waiting for data not yet computed) grows with the number of processor-memory units**
  - **One imagines that efficiency should be bounded between zero and unity, approaching zero asymptotically as overhead and redundant work eventually overwhelms useful work**
  - **In fact, "superunitary efficiency" (often called "superlinear speedup") is common as smaller storage per processor uses memory more efficiently**
- **"Isoefficiency" describes the scaling of work and memory per processor that preserves a constant efficiency of the computation, asymptotically; see, e.g., Grama *et al.*, *Intro. to Par. Comput.* (2003)**

Contraindications of scalability

- **For fixed problem size**
  - **Concurrency limitations with growing overhead**
    - **"fully resolved" discrete problems (protein folding, network problems)**
    - **"sufficiently resolved" problems from the continuum**
- **For scalable problem size**
  - **Resolution-limited progress in "long time" integration**
    - **explicit schemes for time-dependent PDEs**
    - **suboptimal iterative relaxations schemes for equilibrium PDEs**
  - **Nonuniformity of threads**
    - **adaptive schemes that fall out of load balance**
    - **multiphase computations in which the simultaneous load balance of the work in different phases is inconsistent with spatial locality (e.g., particles and fields)**

Amdahl's Law

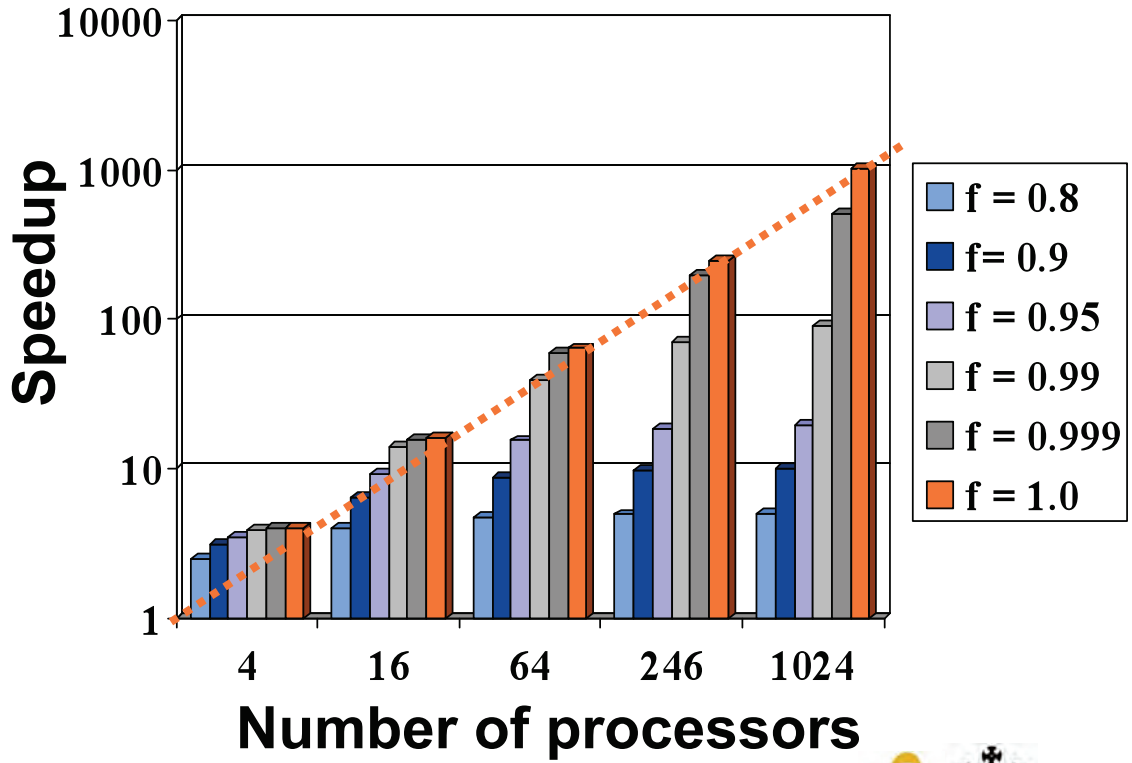
Gene Amdahl of Cray Computer formulated his famous pessimistic formula about the speedup available from concurrency in 1967. If *f* is the fraction of the code that is parallelizable and *P* is the number of processors available, then the time T_P to run on *P* processors as a function of the time T_1 to run on 1 is:

$$T_P = f\frac{T_1}{P} + (1-f)T_1$$

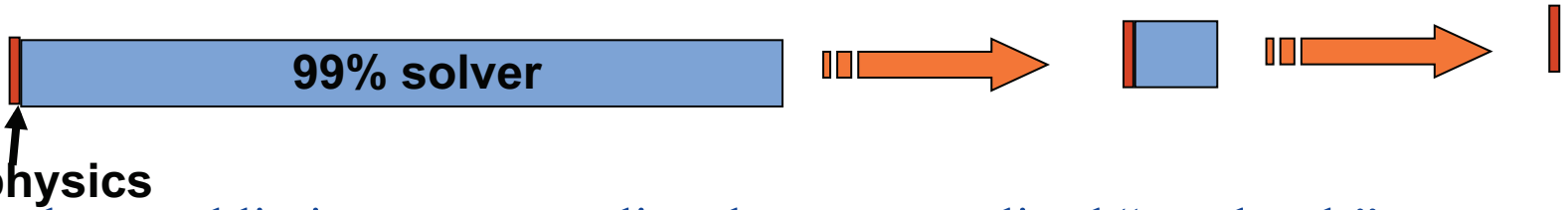
Speedup:

$$\frac{T_1}{T_P} = \frac{1}{(1-f) + f/P}$$

$$\lim_{P\to\infty} Speedup = \frac{1}{(1-f)}$$



Amdahl's Law illustrated



- Fundamental limit to strong scaling due to generalized "overheads"
- Asymptotically, time is independent of processors available
- To analyze: bin code segments by degree of exploitable concurrency and divide by processors available; each bin saturates as P increases
- Illustration for just two bins:
  - fraction f_1 of work that is purely sequential
  - fraction $(1\text{-}f_1)$ of work that is arbitrarily concurrent
- Wall clock time for P processors $\propto f_1 + (1 - f_1)/P$
- Speedup $= 1/[f_1 + (1 - f_1)/P]$
  - for f_1=0.01

P	1	10	100	1000	10000
S	1.0	9.2	50.3	91.0	99.0

- Applies to any performance enhancement, not just parallelism

Memory-constrained scaling under superlinear algorithmic complexity

- Illustrate for CFL-limited time stepping
- Parallel wall clock time

$$\propto T\, S^{1+\alpha/d}\, P^{\alpha/d}$$

- Example: explicit wave problem in 3D ($\alpha=1$, $d=3$)

Domain	$10^3 \times 10^3 \times 10^3$	$10^4 \times 10^4 \times 10^4$	$10^5 \times 10^5 \times 10^5$
Exec. time	1 day	10 days	3 months

- Example: explicit diffusion problem in 2D ($\alpha=2$, $d=2$)

Domain	$10^3 \times 10^3$	$10^4 \times 10^4$	$10^5 \times 10^5$
Exec. time	1 day	3 months	27 years

d-dimensional domain, length scale ***L***

d+1-dimensional space-time, time scale ***T***

h mesh cell size

τ time step size

τ=O(h^α) bound on time step

n=L/h number of mesh cells in each dim

N=n^d number of mesh cells overall

M=T/τ number of time steps overall

O(N) total work to perform one time step

O(MN) total work to solve problem

P number of processors

S storage per processor

PS total storage on all processors ***(=N)***

O(MN/P) parallel wall clock time

$\propto (T/\tau)(PS)/P \propto T\, S^{1+\alpha/d}\, P^{\alpha/d}$

(since $\tau \propto h^\alpha \propto 1/n^\alpha = 1/N^{\alpha/d} = 1/(PS)^{\alpha/d}$)

Where does α come from?

- Exponent α is from the Courant-Friedrichs-Lewy (CFL) stability limit on the size of the time step, $\tau = O(h^{\alpha})$
  - well-known to first-year numerical analysis students
  - not well-known to CS students, who may therefore propose mathematically naïve scalings
- For a hyperbolic problem, $\alpha = 1$
- For a parabolic problem, $\alpha = 2$
- Taking a time step larger than this, relative to the spatial step, will allow errors that creep into the solution to amplify to arbitrarily large values as time evolves, and overwhelm all significant digits in the numerical solution
- We can make $\alpha = 0$ with implicit temporal discretizations

Where does α come from?, cont.

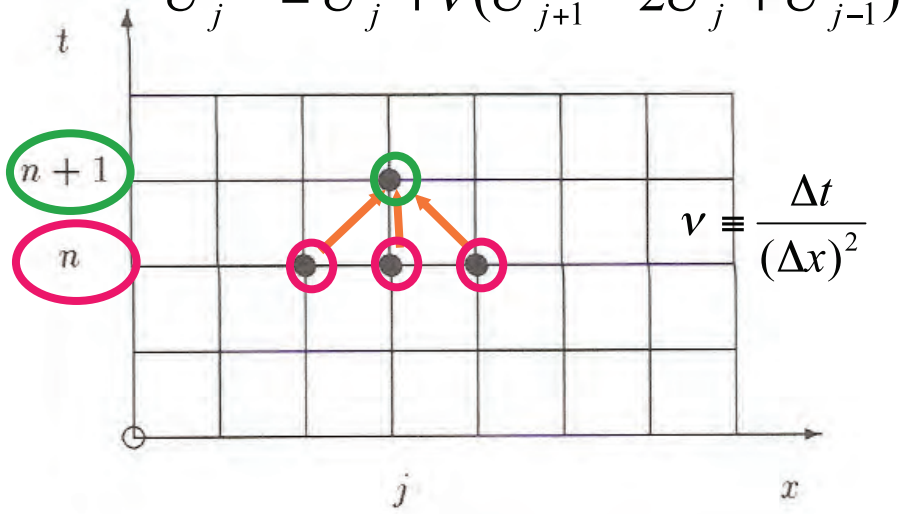
- There are many ways to derive the CFL limits, which were discovered in 1928. For instance, for the parabolic problem
  - *Sufficient* conditions for bounded error are derived as follows
    - Expand a propagator operator for the discrete solution in Fourier modes, and force its norm to be less than unity
    - Consider the weights (as function of h and τ) for formula by which the discrete solution at the next time level is assembled from the solution at the current time level, and force the weights to lie between 0 and 1
  - To see that the resulting conditions are *necessary*
    - Consider one possible discrete error mode – the odd-even grid point oscillation is the most sensitive – and show that it will grow exponentially if the CFL criterion is violated

"Explicit" versus "implicit"

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

- ***Explicit*** **methods evaluate a function of state data at prior time, to update each component of the current state independently**
  - **equivalent to matrix-vector multiplication, in linear problems**

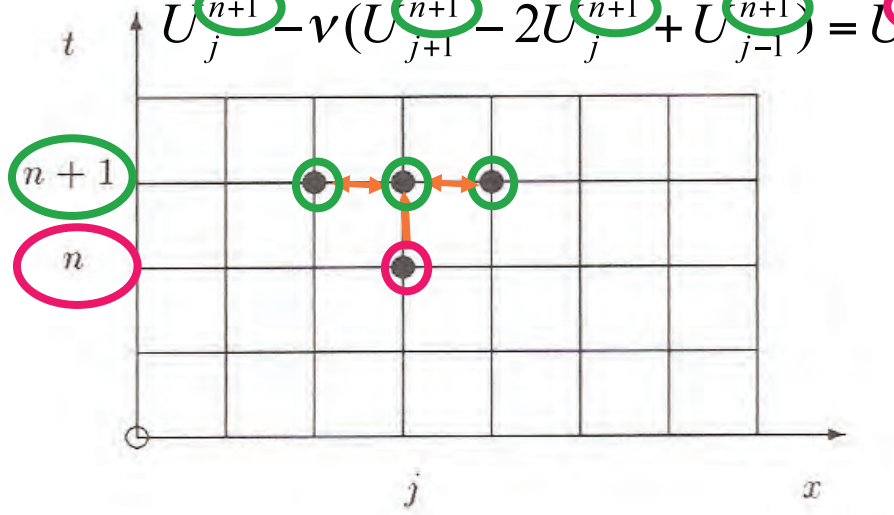
$$U_j^{n+1} = U_j^n + \nu(U_{j+1}^n - 2U_j^n + U_{j-1}^n)$$



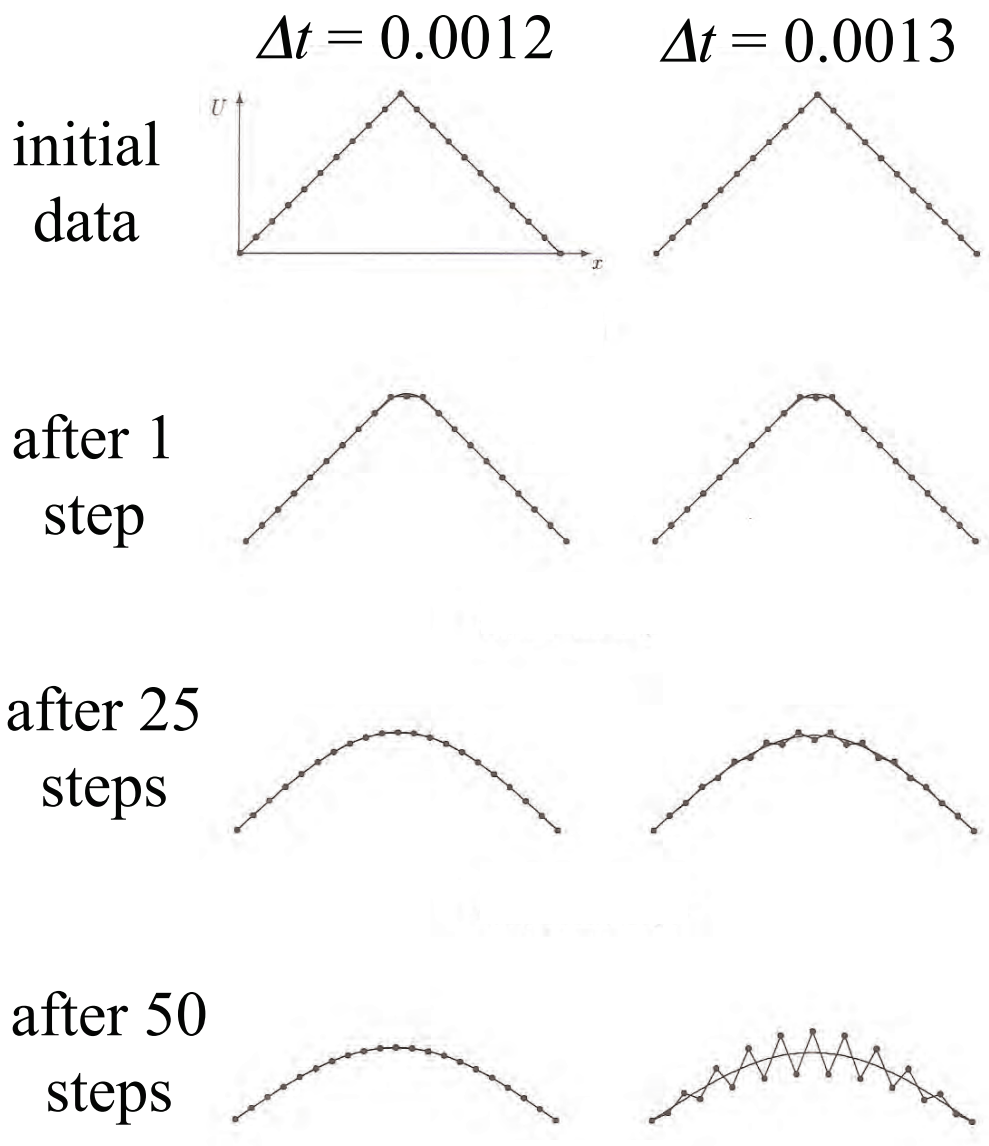
$$\nu \equiv \frac{\Delta t}{(\Delta x)^2}$$

- ***Implicit*** **methods solve a function of state data at the current time, to update all components simultaneously**
  - **equivalent to inverting a matrix, in linear problems**

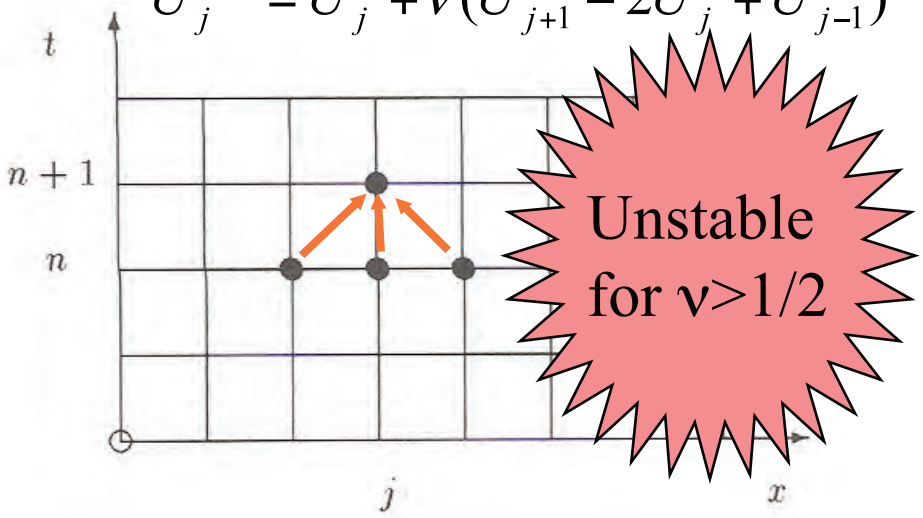
$$U_j^{n+1} - \nu(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}) = U_j^n$$



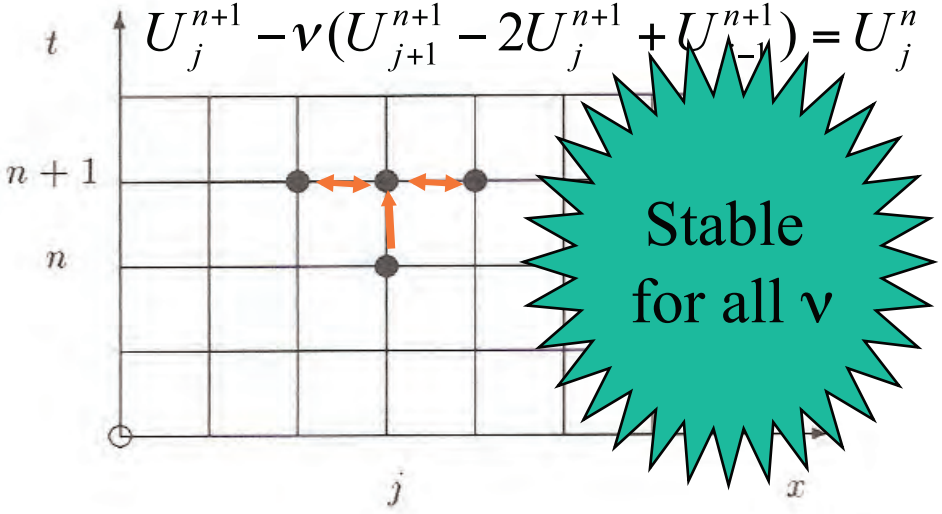
Explicit methods have unforgiving stability boundaries



$$U_j^{n+1} = U_j^n + \nu(U_{j+1}^n - 2U_j^n + U_{j-1}^n)$$



$$U_j^{n+1} - \nu(U_{j+1}^{n+1} - 2U_j^{n+1} + U_{j-1}^{n+1}) = U_j^n$$



c/o K. Morton & D. Mayers, 2005

Algorithmic prototypes: “13 dwarfs”

- **“13 dwarfs” goes back to Colella’s “7 dwarfs” (2004), to which combinatorial “friends” have been added by Asanovic *et al.* of Berkeley (2006)**
- **In their own words:**

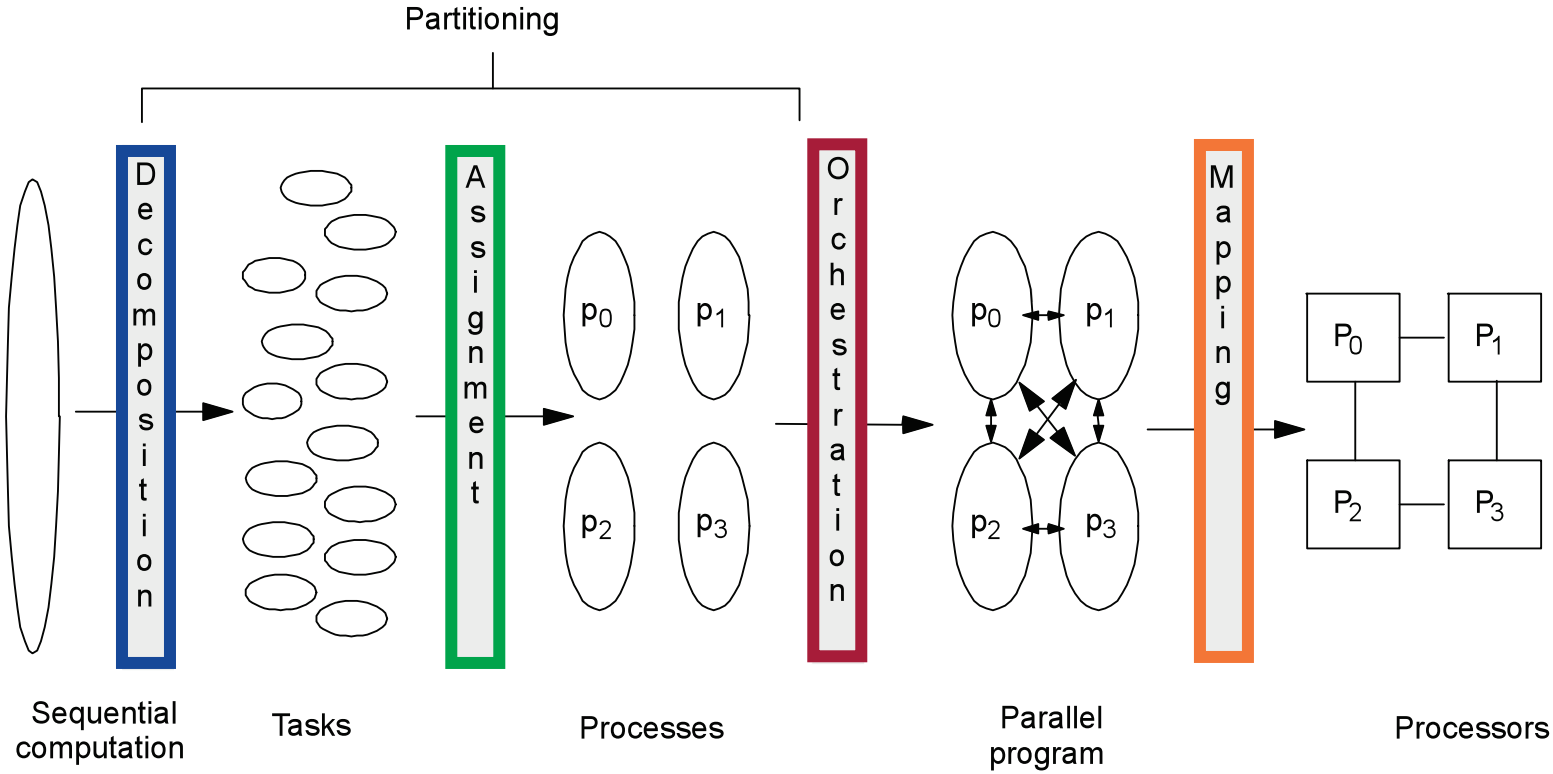
“… the point of the 13 Dwarfs is *not* to identify the low hanging fruit that are highly parallel. The point is to identify the kernels that are the core computation and communication for important applications in the upcoming decade, independent of the amount of parallelism. To develop programming systems and architectures that will run applications of the future as efficiently as possible, we must learn the limitations as well as the opportunities. We note, however, that inefficiency on embarrassingly parallel code could be just as plausible a reason for the failure of a future architecture as weakness on embarrassingly sequential code.”

The 13 algorithmic prototypes*

- Dense direct solvers
- Sparse direct solvers
- Spectral methods
- N-body methods
- Structured grids / iterative solvers
- Unstructured grids / iterative solvers
- Monte Carlo ("MapReduce")
- Combinatorial logic
- Graph traversal
- Graphical models
- Finite state machines
- Dynamic programming
- Backtrack and branch-and-bound

* *The Landscape of Parallel Computing Research: The View from Berkeley*, UCB/EECS-2006-183

Four steps in creating a parallel program



- **Decomposition of computation in tasks**
- **Assignment of tasks to processes**
- **Orchestration of data access, communication, synchronization**
- **Mapping processes to processors**

Dense direct solvers

Description

These are the classic vector and matrix basic linear algebra subroutines (BLAS), traditionally divided into Level 1 (vector/vector), Level 2 (matrix/vector), and Level 3 (matrix/matrix) operations. Data is typically laid out in locally contiguous 1D and 2D arrays.

Uniprocessor Mapping

These kernels are a natural match for vector machines with data accesses mostly having unit and constant strides. For cache based machines, the development of block algorithms that perform N^3 work on just N^2 loads and stores yields performance for Level 3 operations that can approach the peak speed of the machine. Thus, many algorithms have been recast in order to exploit matrix/matrix operations as much as possible.

Parallel Mapping

In addition to memory hierarchy issues, data distribution for load balance becomes a priority here. 2D block cyclic distributions have are preferred for many matrix algorithms. The use of computation/communication overlap has permitted these algorithms to scale well, as exhibited by the tremendous performance of the Top 500 High Performance LINPACK benchmark. The downside is that there are no scientific apps represented legitimately as dense matrices beyond about N=10^4, and dense direct solvers do not have simultaneous time- and memory-scaling.

Sparse direct solvers

Description

Sparse matrix algorithms are used when matrix elements are dominated by zero entries, so that it becomes advantageous, for storage or efficiency reasons, to "squeeze" them out of the matrix representation. Compressed data structures, keeping only the nonzero entries and their indices, are the norm: Compressed Sparse Row (CSR) and Block-Compressed Sparse Row (BCSR). The matrix elements are stored in a single long vector. Each float is accompanied by on average slightly more than one integer that describes its location in the original matrix.

Uniprocessor Mapping

Sparse direct codes are complex, with graph algorithm components that attempt to exploit the structure of the nonzeros. In addition there are a high number of integer operations and indexed accesses. This integer indirection leads to a small percentage of peak floating point operations. Modern algorithms on cache-based machines also attempt to take advantage of dense blocks so that the high-performing Level 3 operations can be used in inner loops.

Parallel Mapping

The sequential dependence structure of many algorithms is intricate. The sparsity implies limited concurrency. In practice, sparse direct solvers for typical PDE applications scale well to hundreds of processors only. However, scientifically relevant problem sizes scale without limit. Like dense solvers, sparse direct solvers are best regarded as a component of a more scalable outer method.

Spectral methods / FFTs

Description

Data is operated on in the spectral domain, transformed from either a temporal or spatial domain. Spectral transformations are typically hierarchical with a number of stages logarithmic in the transform size, where the dependencies within a stage form a set of "butterfly" patterns (e.g., the Cooley-Tukey Fast Fourier Transform or the many variants of Wavelet transforms). Multidimensional transforms are typically composed as the sequential application of 1D transforms.

Uniprocessor Mapping

Codes are highly vectorizable using unit-stride and constant-stride addressing, although efficiency is greater when multiple transforms are performed simultaneously or the problem is large enough that it can be broken into several simultaneous smaller ones. The address pattern is regular and statically determinable so prefetching is effective, although the strides change between stages.

Parallel Mapping

Although the HPC Challenge FFTE benchmark is a parallel 1D spectral transform, partitioning a 1D FFT across multiple processors is rare in practice. Normally, a multidimensional transform is partitioned across multiple processors such that a number of 1D transforms are performed locally on each processor. The algorithms are refactored so that most stages are computed using data local to a node, with a few stages requiring global all-all communication for a transpose operation. For a distributed memory machine, each row of data is contiguous for each processor. The data is then transposed so that the columns contiguous on each processor. Scaling is often excellent.

N-body methods

Description

N-body problems depend on interactions between many discrete point sources. There are many variations: in particle-particle methods, every point depends on all others, leading to an O(N^2) calculation. Hierarchical particle methods combine forces or potentials from multiple points to reduce the computational complexity to O(N log N) for Barnes-Hut or O(N) for Fast Multipole.

Uniprocessor Mapping

The full N^2 algorithm lacks simultaneous time- and memory-scaling. This simple approach iterates through every point, adding forces from every other point with multiple unit-stride accesses through a large array. Cache-blocking is used, but because the points can move, the structures have to be rebalanced occasionally. Divide-and-conquer is better: these mathematically sophisticated strategies treat multiple distant points as a single point and recurse hierarchically. A recursive mapping of the points in this fashion is represented as a tree of nested boxes. Barnes-Hut (BH, 1986) computes the force. Greengard-Rokhlin Fast Multipole (FM, 1987) computes the potential. FM uses a fixed set of boxes with varying information per box, rather BH's varying number of boxes with fixed information.

Parallel Mapping

One of the most important issues with running parallel N-body methods is load balancing (including dynamic rebalancing). As bodies move, the hierarchy that is exploited in the advanced N-body methods becomes more difficult to harness efficiently, but scaling is often excellent.

Structured grids / iterative solvers

Description

Data is arranged in a regular multidimensional grid. Computation proceeds as a sequence of grid updates. At each step, all points are updated using values from a small neighborhood. The updates are logically concurrent, but in practice are implemented as a sequential sweep through the local computational domain. Updates may be in place, overwriting the previous version, or may be performed in parallel, if two copies of the grid are used or updates alternate (e.g., red-black) to avoid read-write conflicts. These codes have a high degree of concurrency, and data access patterns are regular and statically determinable. Patchwise Cartesian adaptive mesh refinement for multiscale problems has logically the same structure on each patch.

Uniprocessor Mapping

Codes are highly vectorizable, using unit-stride or constant-stride memory accesses. The points can be visited in an order (both within one update step and across multiple sequential update steps) that provides spatial locality to make good use of long cache lines and temporal locality to allow cache reuse. Spatial locality is very high, and either hardware or software prefetching is effective given the predictable addressing pattern. Temporal locality within a step is limited, with additional temporal locality across update steps.

Parallel Mapping

Data is decomposed by domain, with ghost cell exchanges to complete cut neighborhoods locally. Deep ghost-cell regions can be used to trade bandwidth to hide additional latency. Parallel efficiency is determined by the surface (communication) to volume (computation) ratio.

Unstructured grids / iterative solvers

Problem

Many problems can be described in the form of updates on an irregular mesh or grid, with each grid element being updated from neighboring grid elements. Considerations of update ordering with multicoloring and partitioning minimizing surface-to-volume ratios mimic those of structured grid, but require graph algorithms. These codes have a high degree of parallelism, but data accesses require indirection, which adds memory for the indexing information and results in poor spatial locality, as illustrated in the following example code:

Uniprocessor Mapping

The code is often highly vectorizable, but requires scatter-gather memory accesses to retrieve data items. An implementation may flatten the data structure so that the neighbors of a given entity or multiple entities may be loaded with a single gather operation. Memory hierarchies are less effective on these codes. Spatial locality is usually limited to one of the entity table types at a time, as the other tables are accessed indirectly with patterns dependent on the mesh structure. There is potential for some temporal locality, as a given entity is used in several different neighborhoods.

Parallel Mapping

The mesh structure is divided into “chunky” contiguous subdomains, where neighboring entities reside together. Communication and synchronization between nodes is mostly between neighbors. The mesh partitioning is usually performed as a preprocessing step, but adaptive algorithms require embedded mesh partitioners that redistribute in response to load imbalances that emerge at runtime.

Monte Carlo ("MapReduce")

Description

This dwarf was originally called "Monte Carlo", after the technique of using statistical methods based on repeated random trials. The patterns defined by the programming model MapReduce are a more general version of the same idea: repeated independent execution of a function, with results aggregated at the end. The defining characteristic is that essentially no communication is required between processes except at the point of aggregation of independently generated results.

Uniprocessor Mapping

MapReduce techniques constitute a general approach that can be applied to a wide variety of problems. The pattern of computation used will depend on the problem being run.

Parallel Mapping

MapReduce techniques are, in general, embarrassingly parallel: since the overall problem involves a number of independent trials, no (or very little) communication overhead is required. The only potential difficulty is that the trials might require different amounts of computation, meaning that different threads of computation might complete at different times. Hence there is the potential for load-balance inefficiency for a single set of merged trials, which can be ameliorated by launching multiple sets of trials.

Combinatorial logic

Description

Combinational logic describes many simple, yet important functions that exploit bit-level parallelism to achieve high throughput. Workloads dominated by combinational logic computations generally involve performing simple operations on very large amounts of data. For example, computing checksums and cyclic redundancy checks (or CRCs) is critical to network processing and ensuring data archive integrity, and can be accelerated with combinatorial logic. Performing simple Boolean operations such as AND, OR, and XOR on large data sets is important for applications such as RAID, and falls into the combinational logic category. Population count, or finding the number of '1's in a word, is also an important combinational logic function useful for data mining. In general, whenever there is bit-level parallelism to be found, the computation can be accelerated by using dedicated combinational logic resources.

Uniprocessor Mapping

On many architectures, the difficulty of programming combinational logic algorithms is due to lack of support for bit-level operations or variable-word-size operations in the instruction set or the programming language.

Parallel Mapping

On many applications that employ combinational logic, algorithms may be broken into data pipelines, where each processor executes part of the pipeline and then passes the data to the next processor.

Graph traversal

Description

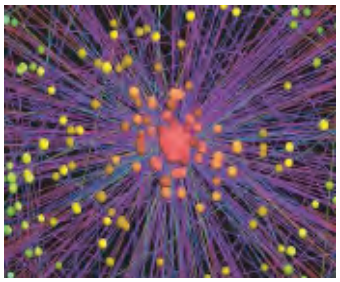
Graph Traversal applications must traverse a number of objects and examine characteristics of those objects. Applications typically involve indirect lookups and little computation.

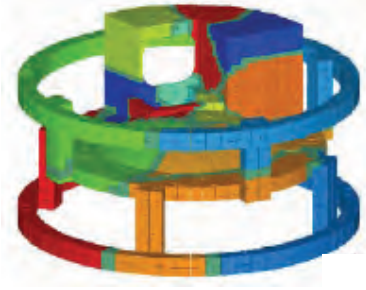
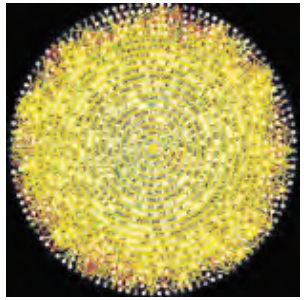
Uniprocessor Mapping

An algorithm that requires little processing and sequential access of successive graph nodes may be equivalent to pointer chasing, without much chance for more efficient processing. Often, though, the reality is better: there may be locality in accesses to the graph, or there might be some processing per node that can reduce the effective cost of finding later nodes.

Parallel Mapping

If the graph can be reproduced on each processor-memory element, independent queries are embarrassingly parallel. If the graph must be distributed over many elements, a single query can be executed in parallel with the possibility of termination in greater or fewer overall link traversals than in serial. Results are highly dependent upon graph structure and graph structures encountered in practice have huge varieties of structure, as measured by average degree or average distance.



Graphical models

Description

A graphical model is a graph in which nodes represent variables, and edges represent conditional probabilities. Graphical models include Bayesian networks (also known as belief networks, probabilistic networks, causal network, and knowledge maps). Hidden Markov models and neural networks are also graphical models.

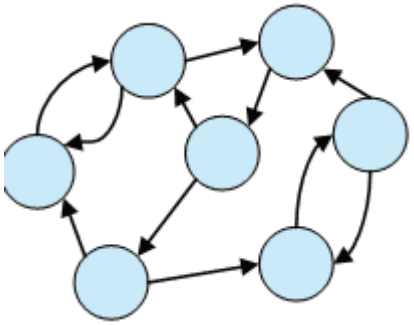
Uniprocessor Mapping

Because graphical models are graphs, evaluating graphical models is a form of graph traversal. Graphical models include a probabilistic aspect that requires a small amount of computation per visited node, but the basic processing is otherwise similar. Constructing graphical models usually involves processing many observations and variables. As each observation is processed, the relevant variables within the model are updated.

Parallel Mapping

In practice, a single graphical model may be evaluated many times for a single problem, or many graphical models can be evaluated for a single input. For example, in speech recognition, sound may be broken into frames of discrete, short segments of time; each frame may be evaluated against many models to derive a distribution of probabilities that the frame matches particular phonemes. Because independent graphical models or inputs can be evaluated independently, a graphical model may be able to be parallelize very simply. Parallel construction of graphical models can be complicated by the possibility of conflicts in the part of the graph that is being updated.

Finite state machines



Description

A finite state machine (FSM), or finite state automaton, is a model of behavior composed of a finite number of states, transitions between those states, and actions. Computation represented as a finite state machine (FSM) is described by an interconnected set of states with transitions between them. A parallel computer can, *itself*, be modeled as a finite state machine.

Uniprocessor Mapping

FSMs are naturally expressed in code as a case statement. Each case represents a different state and the switch statement evaluates the current state and the input conditions. The general form of implementing a FSM on a uniprocessor takes the following form. Each f may produce output and must compute a single next state based solely on the input states (i.e., f must not contain any state of its own). Each time through the case statement represents another state transition.

Parallel Mapping

Parallelism in FSMs is often difficult to utilize. Since there is only one state active at any one time, only one thread of execution is naturally capable of executing it. However, opportunities come either from decomposition or composition with other finite FSMs. A processing element would calculate its FSM's next state based on the original inputs, its current state, and outputs from other state machines it is interacting with. Decomposition ideally creates smaller simpler state machines dividing the next state and output computation across processing elements. Parallelism exploited must justify the communication overhead introduced.

Dynamic programming

Description

Dynamic programming techniques are used in a variety of industrial size problems, such as VLSI design and sequence matching of DNA strands. The knapsack problem is a classic: given a set of N books and a bag. Each book has an integer weight and an integer profit, and the bag has an integer capacity, C. The objective is to choose a subset of the books whose total weight is no greater than the bag capacity while maximizing the profit. Dynamic programming can be used to heuristically solve this problem in pseudo-polynomial time.

Uniprocessor Implementation

Many of dynamic programming base algorithms assume a fast store and look up time for subproblem solutions. This assumption is challenged by memory hierarchies. For structured dynamic programming applications, the data access pattern is highly regular. Many of the techniques for structured grid, such as special subproblem visit ordering to enhance data spatial locality and temporal locality (i.e., caching) can be applied. The optimization of execution order, though just as applicable to the unstructured dynamic programming applications, is more difficult.

Parallel Implementation

The subproblems in dynamic programming are overlapping by nature. If each overlapping subproblem is required to be solved only once, the overlap creates a data dependency. In a parallel implementation, the communication vs. computation trade-off determines how the subproblems can be solved. A subproblem could be solved once and the solution communicated, or it could be resolved on multiple PEs to reduce communication cost.

Backtrack and branch-and-bound

Description

Branch-and-bound algorithms are effective for solving various search and global optimization problems. The goal is to find a globally optimal solution in an intractably large space. Some method is required in order to rule out regions of the search space that contain no interesting solutions. Branch and bound algorithms work by the divide and conquer principle: the search space is subdivided into smaller subregions (this subdivision is referred to as branching), and bounds are found on all the solutions contained in each subregion. The strength of branch and bound comes when bounds on a large subregion show that it contains only inferior solutions, and can be discarded without further examination. The traveling salesman problem is classic: link up N cities whose pairwise distances are known with a tour of minimum length.

Uniprocessor Mapping

The uniprocessor mapping of branch and bound algorithms is straightforward: the processor explores the search space, using problem specific heuristics to guide the search into productive regions of the space, and problem specific bounding methods at each node of the search process.

Parallel Mapping

The search space can be divided such that each processor owns a different subregion, and then each processor can explore independently. In such an approach, dynamic load balancing issues are of crucial importance, since dividing up the search space amongst available processors effectively imposes a certain branching, and it is not possible in general to know a good branching strategy a priori which will keep all processors equally busy.

An up-close look at the scaling of sparse iterative solvers

- In order to make issues of scaling concrete in perhaps the most common kernel in scientific applications, the solution of a large, sparse linear system, we now examine the preconditioned Krylov subspace method – a classic sparse iterative solver – in some detail
- A Krylov subspace method builds up a solution to a linear operator equation, $Ax=b$, by choosing an optimal x within a subspace of increasing dimension, which gets successively closer to solution x^*
- The inner kernel of the Krylov method is multiplication of a vector by the sparse operator A – this is exactly like evaluating a stencil function over the structured or unstructured grid used to discretize a PDE to obtain A
- Preconditioners, including multigrid, have local sparse and dense direct solves as building blocks; hence the preconditioned Krylov method relies on three (and more) of the dwarfs

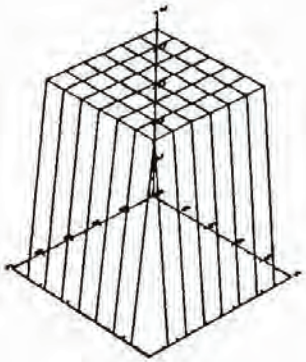
Krylov bases for sparse systems

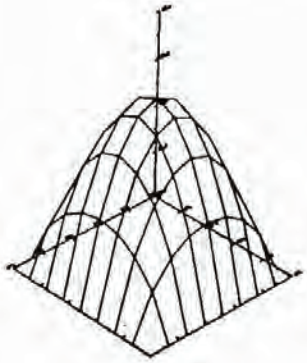
- E.g., conjugate gradients (CG) for symmetric, positive definite systems, and generalized minimal residual (GMRES) for nonsymmetry or indefiniteness
- Krylov iteration is an algebraic projection method for converting a high-dimensional linear system into a lower-dimensional linear system

$$Ax = b \qquad x = Vy \qquad g = W^T b \qquad Hy = g \qquad H \equiv W^T A V$$

Krylov bases for sparse systems

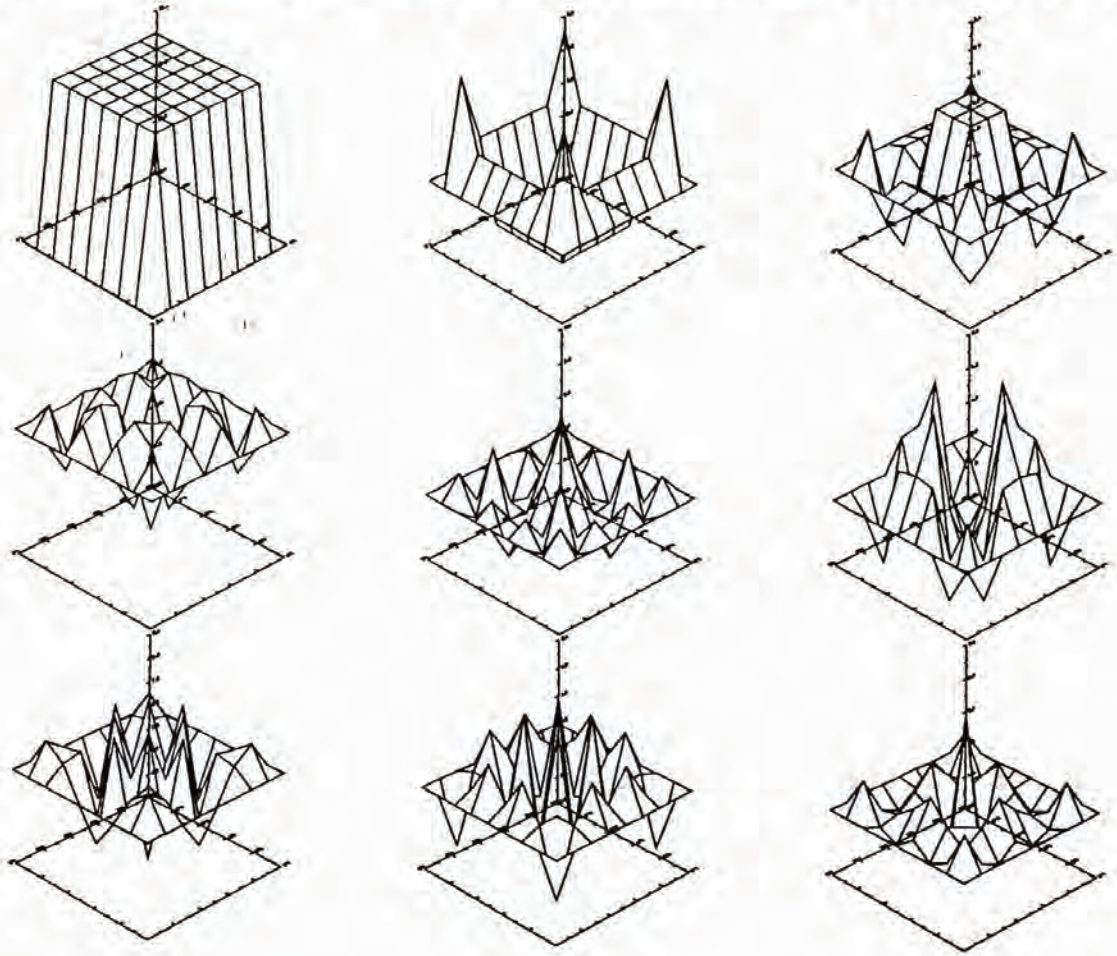
For an easy-to-visualize example of a Krylov basis, GMRES was used to solve Poisson's equation on an 8×8 grid on a unit square with no preconditioning. The right-hand side and the solution were:





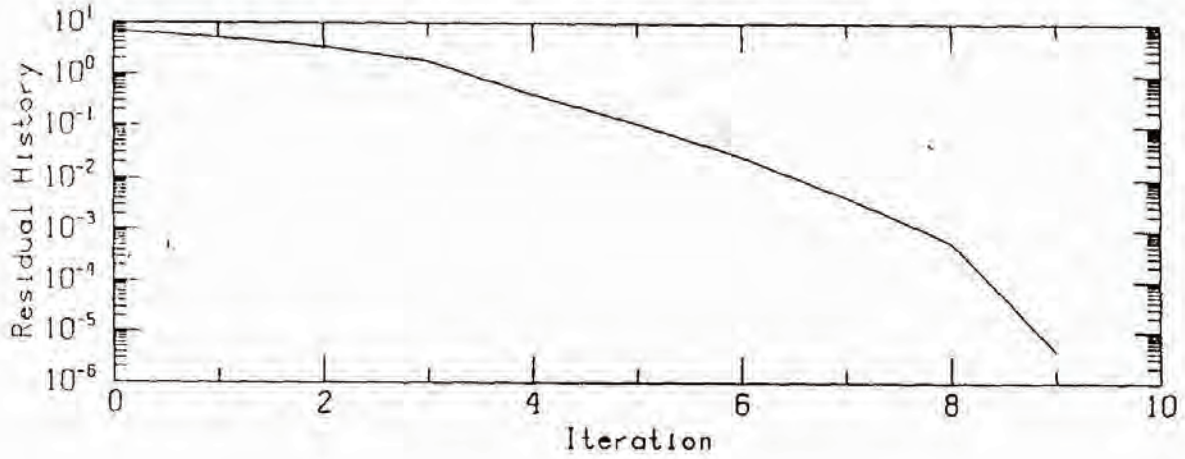
Krylov bases, cont.

GMRES formed the solution as a linear combination of these nine Krylov vectors, in order:

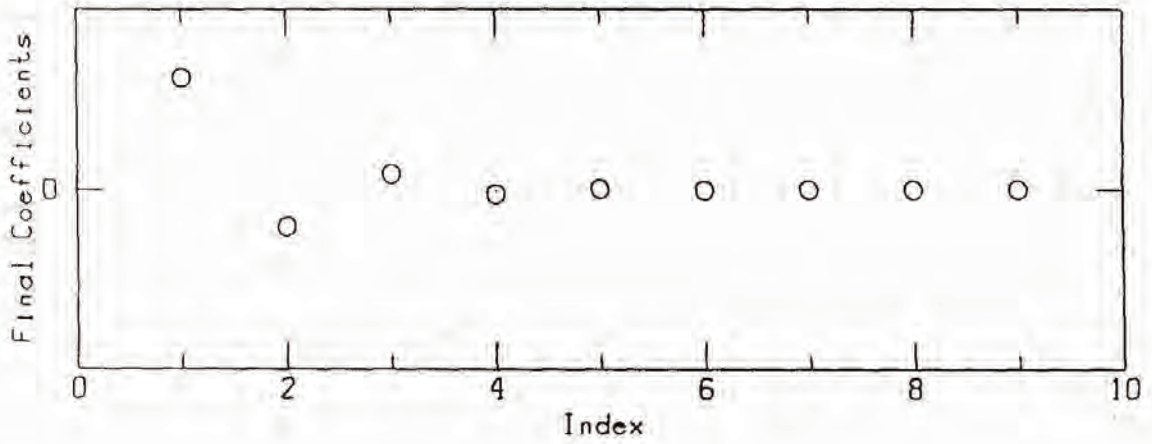


Krylov bases, cont.

The iteration history is:

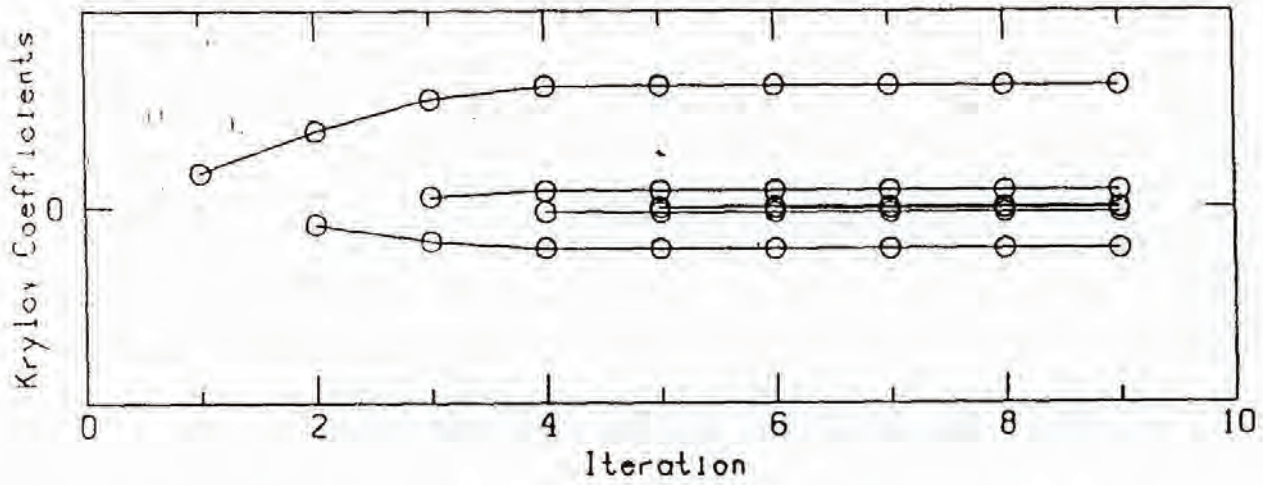


The converged magnitudes of the coefficients are:



Krylov bases, cont.

The iteration history of each coefficient is:



Note that only 9 vectors were needed in this problem of 81 degrees of freedom.

History

- Aleksei N. Krylov (1863-1945) was a naval architect in St. Petersburg
  - Gave formal definition for finding least squares solution to linear system within a subspace
  - Method was not made practical in his lifetime
- Method of Conjugate Gradients (CG) invented in 1950 (published 1952) as a direct method by Hestenes and Stiefel
- Reinvented in 1971 by Reid as an iterative method
- IC-CG popularized in 1977
- 1980s and 1990s full of discovery and invention

Key features

- Access linear operator by matrix-vector operations only
  - Av or $A^t v$ or both, per iteration
- Ideal for sparse operators
- Ideal for operators that are not explicitly stored (action evaluated by subprogram)
- Ideal for parallelism when transpose is not needed and matrix can be partitioned in block rows

Limitations

- Loss of orthogonality/conjugacy in finite precision arithmetic
- Reliance on frequent global inner products
  - unlike Chebyshev methods
  - induces synchronization
- For nonsymmetric systems, either:
  - lack of strong convergence results for low-storage versions, *or*
  - requirement to store many (dense) vectors (recurrence relation truncates only for symmetric case)

Advantages

- No need to estimate spectra (eigenvalues) and corresponding eigenvectors of the matrix
  - unlike Chebyshev methods
- Low cost, general-purpose "wrap" around generally more sophisticated or expensive preconditioner

Interesting research frontiers

- Early termination of Krylov iterations
- Reuse of Krylov subspaces from one system to the next
  - nearby right-hand sides
  - nearby matrix operators
- Blocking of multiple matrix-vector multiplies between calls to inner product routines
  - better use of cached matrix elements for higher serial performance
  - less frequent global synchronization for higher parallel performance

Key convergence estimates

Symmetric case: $Ax = f$.

Theorem. If A has at most s distinct eigenvalues, CG converges in at most s steps.

Theorem. Let $e_k \equiv x_k - x^*$, where x^* is the exact solution, and let $\kappa \equiv \lambda_{\max}(A)/\lambda_{\min}(A)$. Then

$$||e_k||_A \leq 2\left(\frac{1-1/\sqrt{\kappa}}{1+1/\sqrt{\kappa}}\right)^k ||e_0||_A \,.$$

Nonsymmetric case: $Ax = f$.

Theorem. If A has at most s distinct eigenvalues, GMRES converges in at most s steps.

Theorem. Let $r_k \equiv f - Ax_k$, $C_p^2 = \lambda_{\max}(A^T A)$, and $c_p = \lambda_{\min}(\frac{1}{2}(A^T + A))$. Then

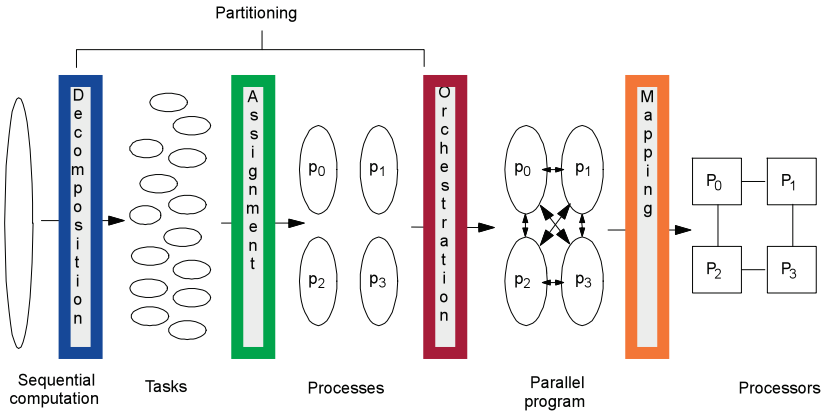
$$||r_k|| \leq \left(1 - \frac{c_p^2}{C_p^2}\right)^{k/2} ||r_0|| \,.$$

Krylov solver software

- There is a large variety of Krylov solvers
  - customized to mathematical properties like symmetry or definiteness
  - customized to architectural properties like synchronization cost, memory capacity, communication to computation capabilities
- All Krylov solvers users have found practical are selectable at runtime in libraries like PETSc and Trilinos
  - users may register their own Krylov solvers if their favorite variant is missing
- Krylov-Schwarz is the most common parallel generalization
  - in the parallel PDE context, the vector space in which the Krylov method searches corresponds to a grid function
  - partitioning the grid imposes a partitioning of the vector space for distributed memory parallelization

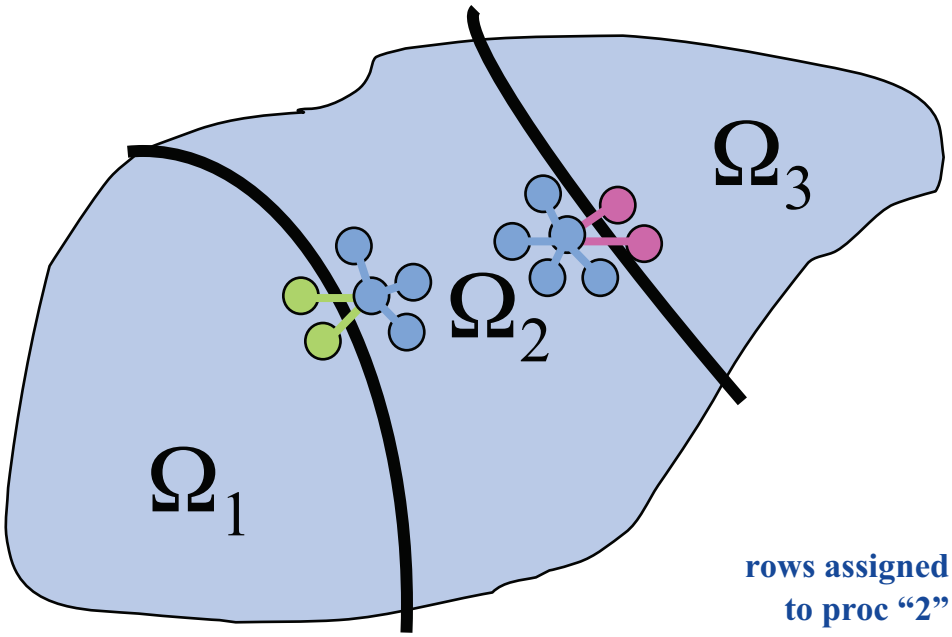
Krylov-Schwarz parallelization is simple!

- Decomposition into concurrent tasks
  - by domain
- Assignment of tasks to processes
  - typically one subdomain per process

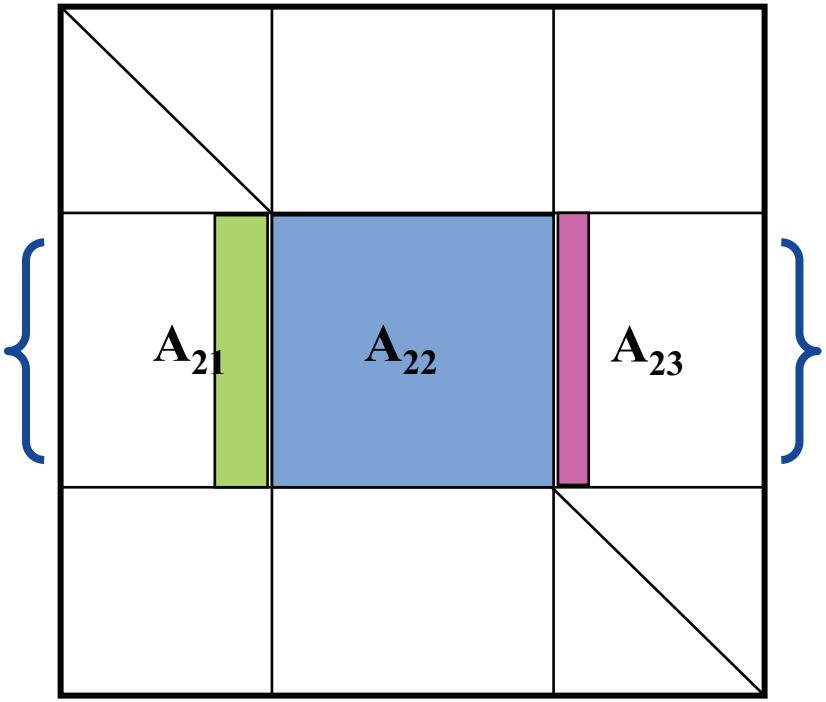


- Orchestration of communication between processes
  - to perform sparse matvec – *near neighbor communication*
  - to perform subdomain solve – *nothing*
  - to build Krylov basis – *global inner products*
  - to construct best fit solution – *global sparse solve (redundantly?)*
- Mapping of processes to processors
  - typically one process per processor

Schwarz-type domain decomposition preserves fixed overhead ratio in memory scaling



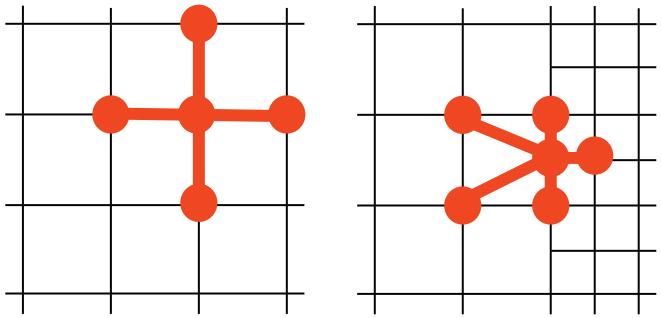
rows assigned
to proc "2"



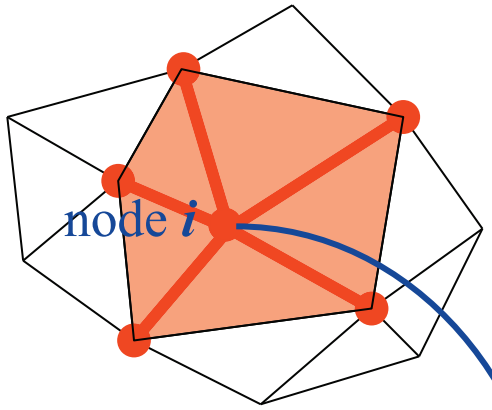
Partitioning of the grid induces block structure on the system matrix (Jacobian)

Schwarz-type domain decomposition is relevant to any local stencil formulation

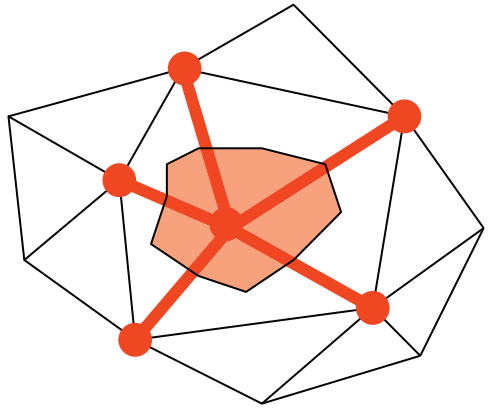
finite differences



finite elements

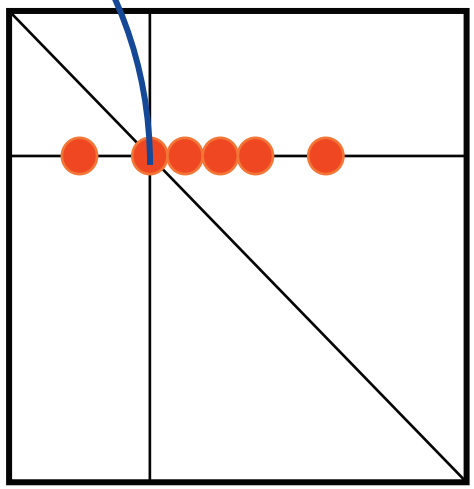


finite volumes



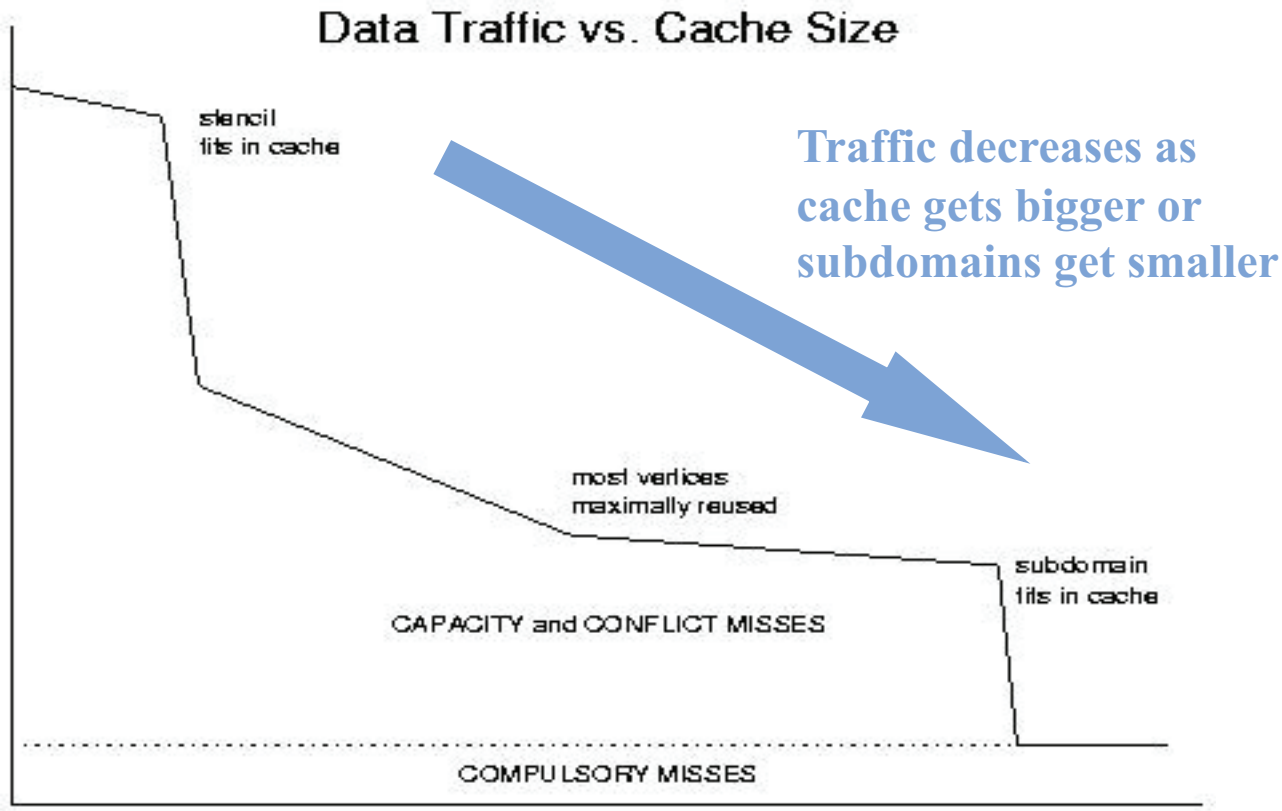
- **lead to sparse Jacobian matrices**
- **however, the inverses are generally dense; even the factors suffer unacceptable fill-in in 3D**
- **want to solve in subdomains only, and use to precondition full sparse problem**

J=



Krylov-Schwarz compelling in serial, too ☺

- As successive workingsets "drop" into a level of memory, capacity (and with effort conflict) misses disappear, leaving only compulsory misses, reducing demand on main memory bandwidth
- Cache size is not easily manipulated, but domain size *is*

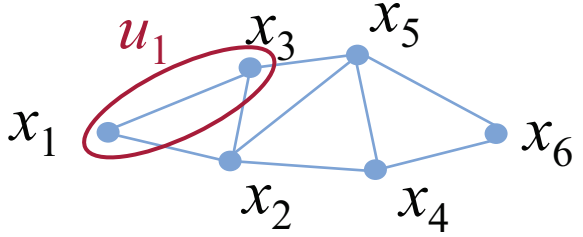


Preconditioned Krylov is simple!

- The original system is $Ax=b$
- We feed to the Krylov solver the equivalent system $(B^{-1}A)x=(B^{-1}b)$
- The action of B^{-1} is supposed to approximate that of A^{-1}, so that the $B^{-1}A$ approximates the identity and the Krylov iteration converges quickly
- The action of B^{-1} is supposed to be cheap
- The algorithm is "optimal" if
  - the cost of multiplication by A is $O(N)$
  - the cost of the action of B^{-1} is $O(N)$
  - the number of Krylov vectors is small, independent of N
- …and parallel optimal if the parallel overhead is at worst $log(N)$

Digression for notation's sake

- We need a convenient notation for mapping vectors (representing discrete samples of a continuous field) from full domain to subdomain and back



$$R_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

- Let R_i be a Boolean operator that extracts the elements of the i^{th} subdomain from the global vector

$$R_1 u = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_3 \end{pmatrix} \equiv u_1$$

- Then R_i^T maps the elements of the i^{th} subdomain back into the global vector, padding with zeros

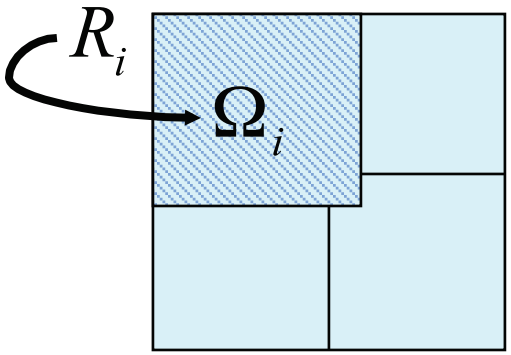
$$R_1^T = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad R_1^T u_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{pmatrix} x_1 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 \\ 0 \\ x_3 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

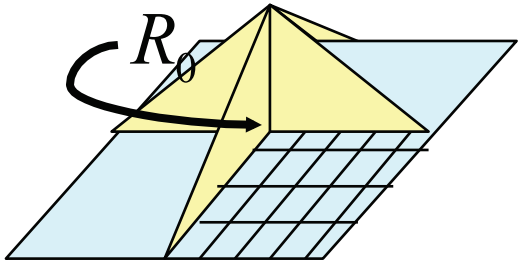
Schwarz domain decomposition method

- Consider restriction and extension operators for subdomains, R_i, R_i^T, and for possible coarse grid, R_0, R_0^T
- Replace discretized $Au = f$ with

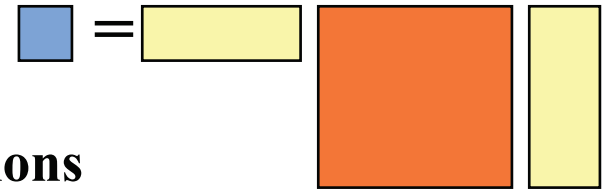
$$B^{-1}Au = B^{-1}f$$

$$B^{-1} = R_0^T A_0^{-1} R_0 + \sum_i R_i^T A_i^{-1} R_i$$





$$A_i = R_i A R_i^T$$



- Solve by a Krylov method
- Matrix-vector multiplies with
  - parallelism on each subdomain
  - nearest-neighbor exchanges, global reductions
  - possible small global system (not needed for parabolic case)

Schwarz formula (projections)

- If A is an operator on a space $\boldsymbol{V}$ and $\boldsymbol{R_i}$ are restrictions into (possibly overlapping) subspaces of $\boldsymbol{V}$, $\boldsymbol{V_i}$, such that $\boldsymbol{V}=\cup\boldsymbol{V_i}$
- Then for a good approximation, $\boldsymbol{B^{-1}}$, to $\boldsymbol{A^{-1}}$:

$$B^{-1} = \sum_i R_i^T (R_i A R_i^T)^{-1} R_i + R_0^T (R_0 A R_0^T)^{-1} R_0$$

or

$$B^{-1} = \sum_i R_i^T A_i^{-1} R_i + R_0^T A_0^{-1} R_0$$

- Then $\kappa(B^{-1}A) = C$

  where C is independent of H and h (resp. P and N)

Comments on the Schwarz results

- **Original basic Schwarz estimates were for:**
  - ***self-adjoint*** **elliptic operators**
  - ***positive definite*** **operators**
  - ***exact*** **subdomain solves,** A_i^{-1}
  - ***two-way*** **overlapping with** R_i, R_i^T
  - ***generous*** **overlap,** $\delta = O(H)$ **(original 2-level result was** $O(1+H/\delta)$**)**
- **Subsequently extended to (within limits):**
  - ***nonself-adjointness*** **(e.g, convection)**
  - ***indefiniteness*** **(e.g., wave Helmholtz)**
  - ***inexact*** **subdomain solves**
  - ***one-way*** **overlap communication ("restricted additive Schwarz")**
  - ***small*** **overlap**

Comments on the Schwarz results, cont.

- Theory still requires "sufficiently fine" coarse mesh
  - However, coarse space need *not* be nested in the fine space or in the decomposition into subdomains
- Practice is better than one has any right to expect

> "In theory, theory and practice are the same ...
> In practice they're not!"
> — *Yogi Berra*

- Wave Helmholtz (e.g., acoustics) is delicate at high frequency:
  - standard Schwarz Dirichlet boundary conditions can lead to undamped resonances within subdomains, $u_\Gamma = 0$
  - remedy involves Robin-type transmission boundary conditions on subdomain boundaries, $(u + \alpha\ \partial u / \partial n)_\Gamma = 0$

"Unreasonable effectiveness" of Schwarz

- When does the sum of partial inverses equal the inverse of the sums? When the decomposition is right!

  Let $\{r_i\}$ be a complete set of orthonormal row eigenvectors for A : $r_i A = a_i r_i$ or $a_i = r_i A r_i^T$

  Then

$$A = \Sigma_i r_i^T a_i r_i$$

  and

$$A^{-1} = \Sigma_i r_i^T a_i^{-1} r_i = \Sigma_i r_i^T (r_i A r_i^T)^{-1} r_i$$

  — the Schwarz formula!

- Good decompositions are a compromise between conditioning and parallel complexity, in practice

Operator projection ideas not limited to Schwarz-style domain decomposition

- In the abstract, Schwarz theory is about polynomials of projection operators
- Appropriate for other types of preconditioners, too
- Suppose we have two preconditioners, each of which is effective on part of the problem, and we use them sequentially

$$u \leftarrow u + B_1^{-1}(f - Au)$$

$$u \leftarrow u + B_2^{-1}(f - Au)$$

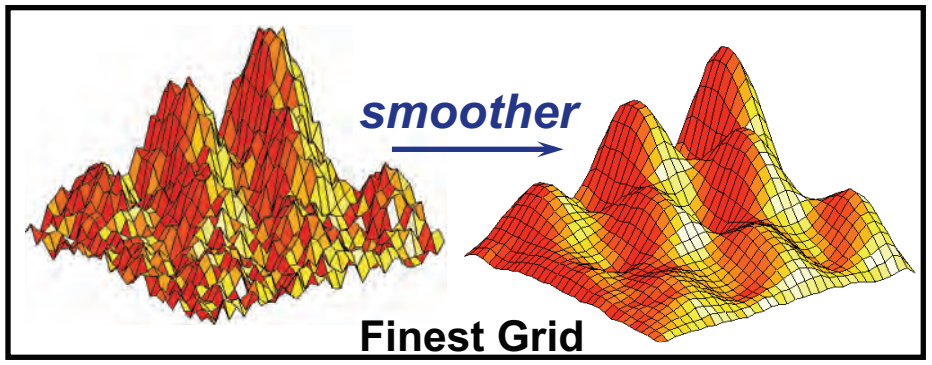
- This leads to a multiplicative scheme:

$$B^{-1} = B_2^{-1} + (I - B_2^{-1}A)B_1^{-1}$$

- This is the form of a standard two-level multigrid scheme in which $\boldsymbol{B_1}$ is a "smoother" and $\boldsymbol{B_2}$ handles the complementary modes:

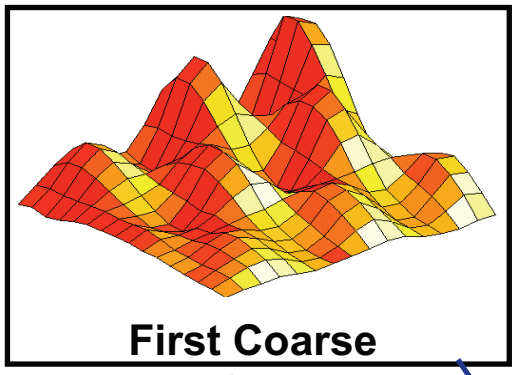
$$B_2^{-1} = R^T A_C^{-1} R\,; \quad A_C = RAR^T$$

Multilevel projection methods



A Multigrid V-cycle

Restriction
transfer from fine to coarse grid



coarser grid has fewer cells (less work & storage)

Recursively apply this idea until we have an easy problem to solve

Prolongation
transfer from coarse to fine grid

Why optimal algorithms?

- The more powerful the computer, the *greater* the importance of optimality
  - though the counter argument is often naively employed ☹
- Example:
  - Suppose *Alg1* solves a problem in time $C\,N^2$, where N is the input size
  - Suppose *Alg2* solves the same problem in time $C\,N\log_2 N$
  - Suppose *Alg1* and *Alg2* parallelize *perfectly* on a machine of 1,000,000 processors
- In constant time (compared to serial), *Alg1* can run a problem 1,000 X larger – memory and time scaling are very different – whereas *Alg2* can run a problem nearly 65,000 X larger

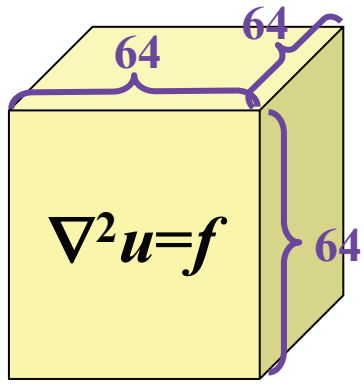
There is no "scalable" without "optimal"

- "Optimal" for a theoretical numerical analyst means a method whose floating point complexity grows at most linearly in the data of the problem, N, or (more practically and almost as good) linearly times a polylog term
- For iterative methods, this means that the product of the *cost per iteration* and the *number of iterations* must be $O(N \log^p N)$
- Cost per iteration must include communication cost as processor count increases in weak scaling, $P \propto N$
  - BlueGene, for instance, permits this with its log-diameter hardware global reduction
- Number of iterations comes from condition number for linear iterative methods; Newton's superlinear convergence is important for nonlinear iterations

Solvers evolve underneath "*Ax=b*"

- Advances in algorithmic efficiency rival advances in hardware architecture
- Consider Poisson's equation on a cube of size $N=n^3$

Year	*Method*	*Reference*	*Storage*	*Flops*
1947	GE (banded)	Von Neumann & Goldstine	n^5	n^7
1950	Optimal SOR	Young	n^3	$n^4 \log n$
1971	CG	Reid	n^3	$n^{3.5} \log n$
1984	Full MG	Brandt	n^3	n^3



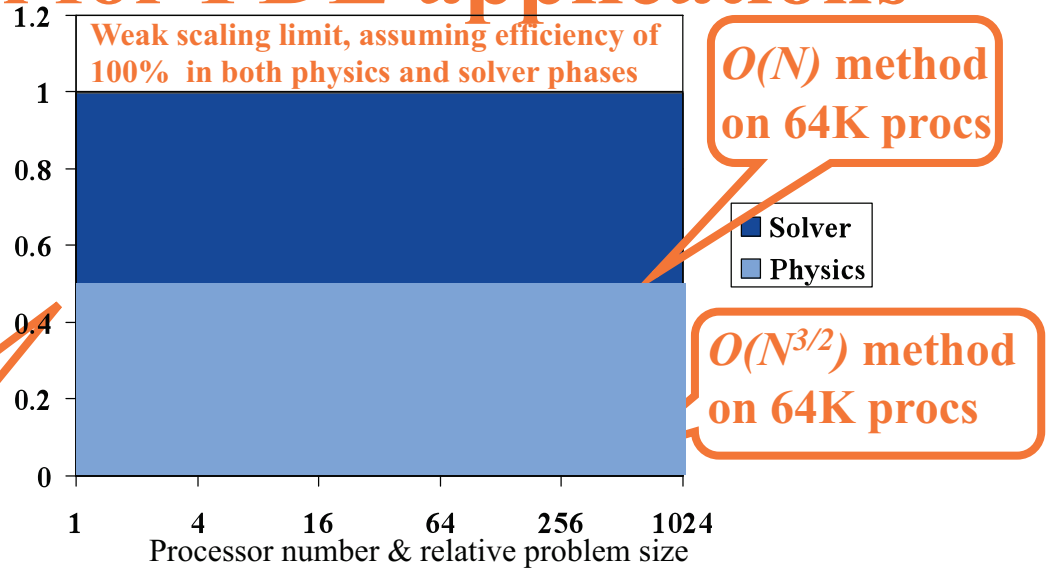
- If $n=64$, this implies an overall reduction in flops of ~16 million*

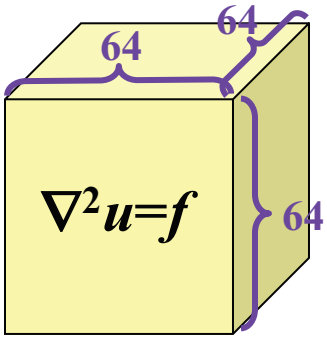
*Six months is reduced to 1 second

Only with optimal complexity solvers are scalable architectures practical for PDE applications

Given, for example:

- a "physics" phase that scales as $O(N)$
- a "solver" phase that scales as $O(N^{3/2})$
- computation is almost all solver after several doublings
- Optimal $O(N)$ solver saves the computational cycles for the physics



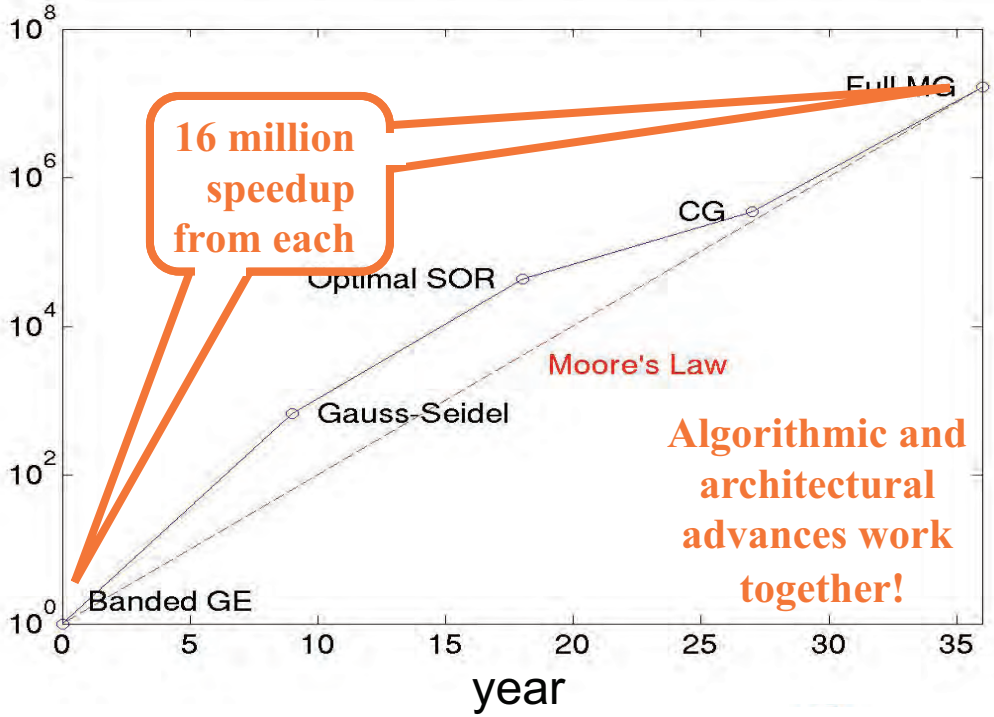


Consider, for example:

- Poisson's equation in a 3D domain
- Solve by "best method available" over a span of 1948 to 1984 (36 years)

Compare with Moore's Law:

- Over 36 years, processor architecture goes through 24 "doubling periods"
- Algorithms produce an equal factor of speedup on a small problem; much *more* on a larger problem

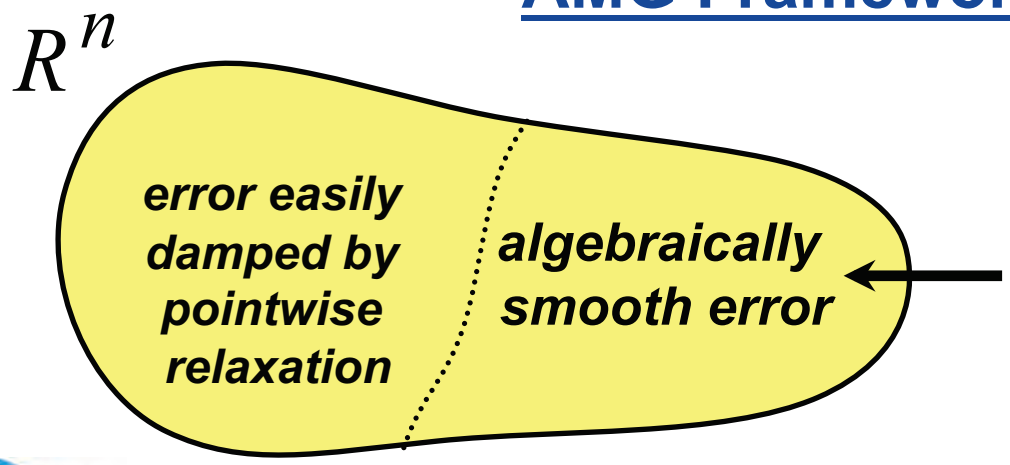


DOE CSGF

Algorithmic optimality: where to go past *O(N)* ?

- Since ***O*(*N*)** is already optimal, there is nowhere further "upward" to go in efficiency, but optimality must extend "outward," to more general problems
- Algebraic multigrid (AMG) has done well with *unstructured*, *inhomogeneous*, and *anisotropic* problems and adaptive AMG (αAMG) has done well with *indefinite*
- More work is needed to obtain ***O*(*N*)** in highly *asymmetric*, *multicomponent* problems and for *high-order alternating sign discretizations*

AMG Framework



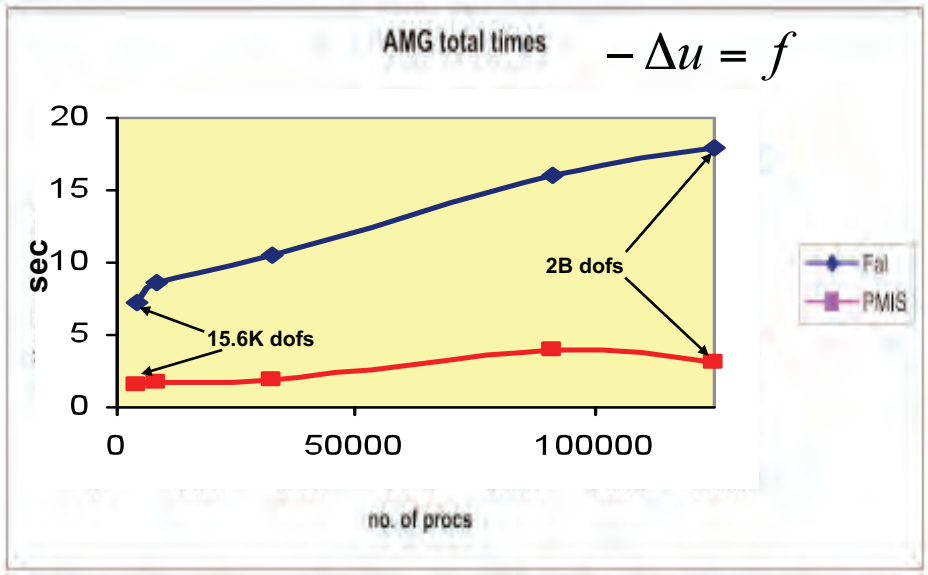
Choose coarse grids, transfer operators, and smoothers to eliminate these "bad" components within a smaller dimensional space, and recur

Algebraic multigrid

- For Poisson, there is a correspondence between the hard-to-smooth error modes and wavenumber, leading to the classification of "fine" (easy to smooth) and "coarse" (hard to smooth, near null-space)
- For more general operators, this *geometrical* correspondence is broken; the "coarse" space is whatever is complementary to the readily smoothable space and is found *algebraically*, in an operator-sensitive way (anisotropy, inhomogeneity, etc.)
- This freedom from geometry is liberating, since problems on unstructured meshes are readily accommodated
- Near null-space modes may now, however, be *dense* to represent, unlike in Poisson (okay, if just a few of them)
- Identifying the coarse space may defy heuristics and need to be found *adaptively* (see **Brezina, *et al.*, SIAM Review 47:317-346 (2005))**

Sparse iterative solvers *are* memory-scaling: algebraic multigrid (AMG) on BG/L (hypre)

- Algebraic multigrid a key and very general algorithmic technology
  - Discrete operator defined for finest grid by the application, itself, *and* for many recursively derived levels with successively fewer degrees of freedom, for solver purposes
  - Unlike geometric multigrid, AMG not restricted to problems with "natural" coarsenings derived from grid alone
- Optimality (cost per cycle) intimately tied to the ability to coarsen aggressively
- Convergence scalability (number of cycles) and parallel efficiency also sensitive to rate of coarsening
- While much research and development remains, multigrid will clearly be practical at BG/P-scale concurrency



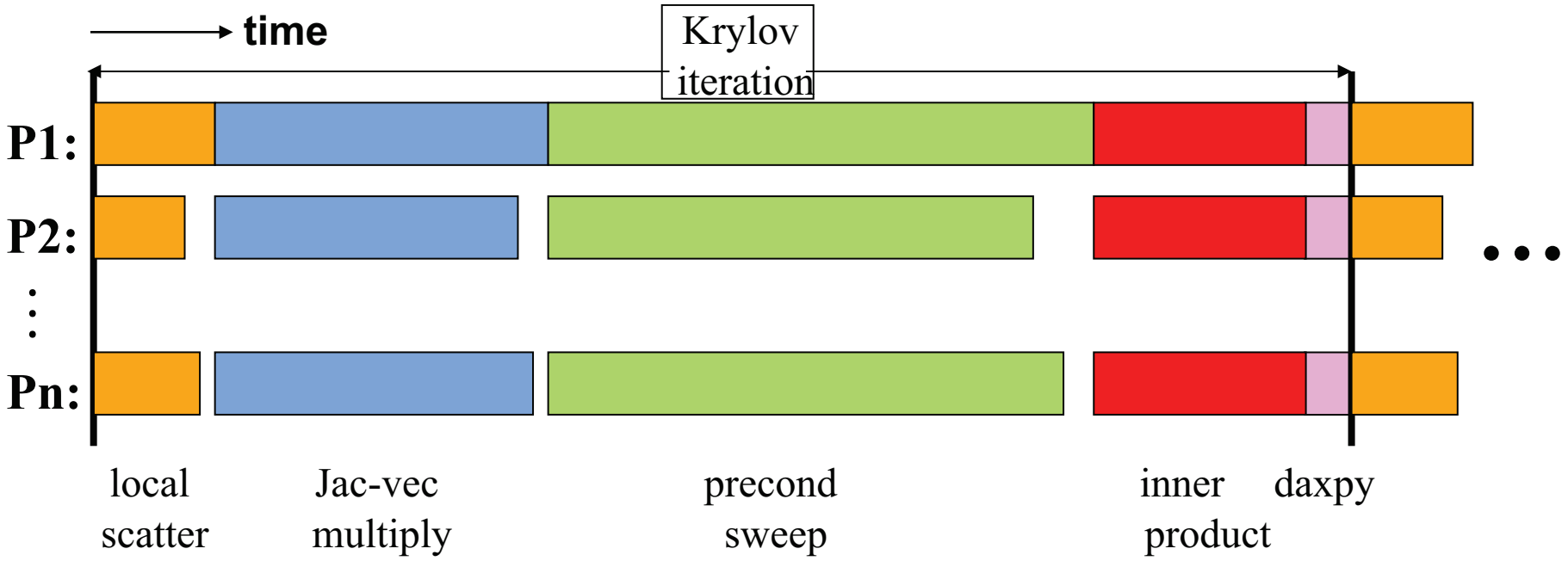
7-pt Laplacian, total execution time, AMG-CG, total problem size ~2 billion

Figure shows weak scaling result for AMG out to 120K processors, with one 25×25×25 block per processor (up to 1.875B dofs)

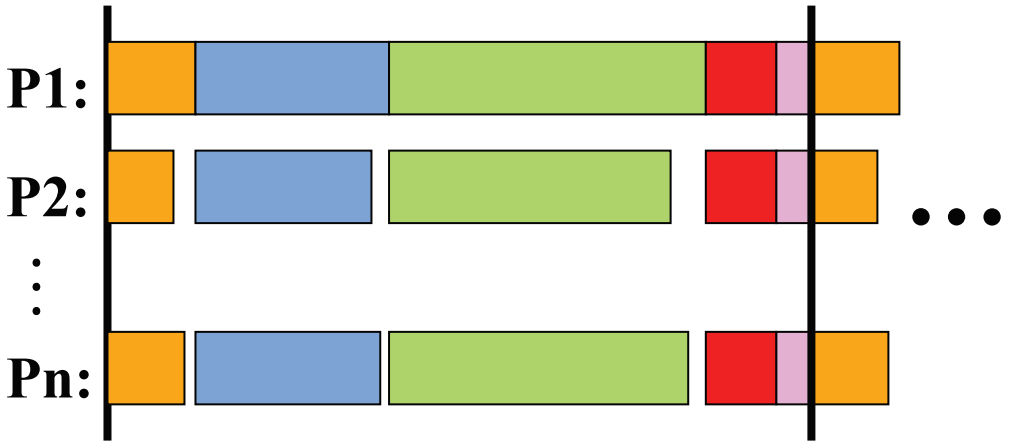
How do solvers weak scale seemingly indefinitely?

- As problem size grows linearly with number of processor-memory elements:
  - maintain constant surface-scaling of communication with volume-scaling of computational work
  - avoid duplicating any data structures that scale with problem size
  - avoid any step with synchronization-induced idleness, except on data sets of constant (therefore asymptotically small) size
  - avoid any step with superlinear growth of complexity in data size, except perhaps logarithmic
  - avoid any iterative method with growth in number of iterations in data size, except perhaps logarithmic
- We know how to do this with sparse systems arising from domain-decomposed finite discretizations of Poisson and parabolic systems
  - adaptation imposes costs to maintain these properties

Inner Krylov-Schwarz kernel in parallel: a Bulk Synchronous Process ("BSP")



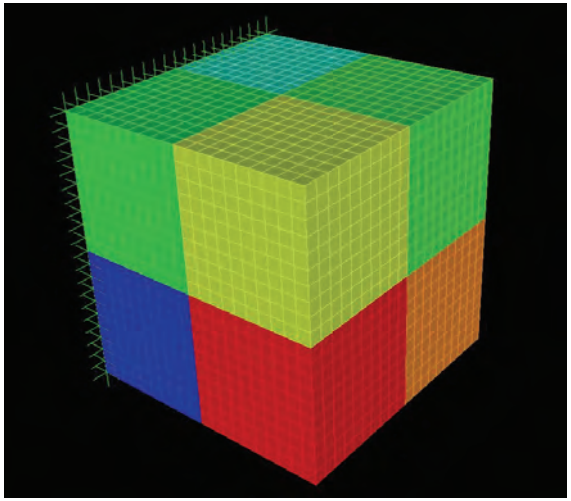
What happens if, for instance, in this (schematicized) iteration, arithmetic speed is ***doubled***, scalar all-gather is ***quartered***, and local scatter is ***cut by one-third***? Each phase is considered separately. Answer is to the right.



Estimating scalability of stencil computations

- Given complexity estimates of the leading terms of:
  - the concurrent computation (per iteration phase)
  - the concurrent communication
  - the synchronization frequency
- And a bulk synchronous model of the architecture including:
  - internode communication (network topology and protocol reflecting horizontal memory structure)
  - on-node computation (effective performance parameters including vertical memory structure)
- One can estimate optimal concurrency and optimal execution time
  - on per-iteration basis, or overall
  - simply differentiate time estimate in terms of *(N,P)* with respect to *P*, equate to zero and solve for *P* in terms of *N*

Estimating 3D stencil costs (per iteration)



- grid points in each direction n, total work $N=O(n^3)$
- processors in each direction p, total procs $P=O(p^3)$
- memory per node requirements $O(N/P)$

- concurrent execution time per iteration $A\, n^3/p^3$
- grid points on side of each processor subdomain n/p
- Concurrent neighbor commun. time per iteration $B\, n^2/p^2$
- cost of global reductions in each iteration $C\, p^{(3/d)}$ or $C' \log p$
  - C includes synchronization frequency
- same dimensionless units for measuring A, B, C
  - e.g., cost of scalar floating point multiply-add

3D stencil computation illustration

Rich local network, tree-based global reductions

- total wall-clock time per iteration

$$T(n,p) = A\frac{n^3}{p^3} + B\frac{n^2}{p^2} + C\log p$$

- for optimal ***p***, $\frac{\partial T}{\partial p} = 0$, or $-3A\frac{n^3}{p^4} - 2B\frac{n^2}{p^3} + \frac{C}{p} = 0,$

or (with $\theta \equiv \frac{32B^3}{243A^2C}$),

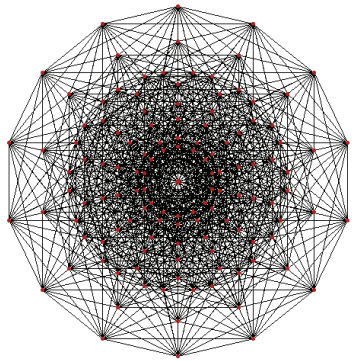
$$p_{opt} = \left(\frac{3A}{2C}\right)^{1/3}\left(\left[1+(1-\sqrt{\theta})\right]^{1/3} + \left[1-(1-\sqrt{\theta})\right]^{1/3}\right)\cdot n$$

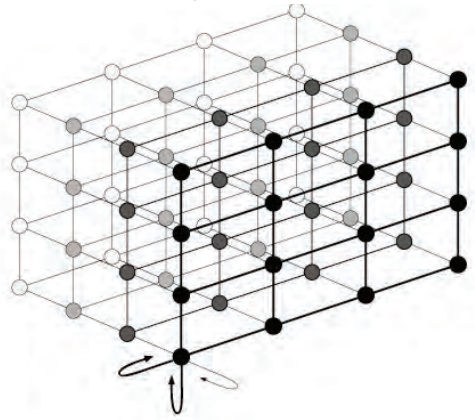
- without "speeddown," ***p*** can grow with ***n***
- in the limit as $B/C \to 0$

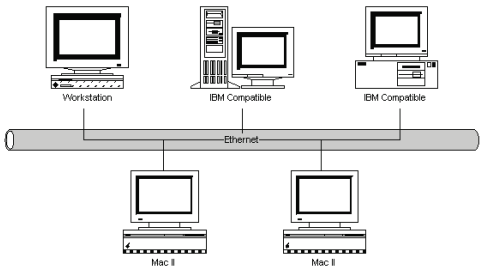
$$p_{opt} = \left(\frac{3A}{C}\right)^{1/3}\cdot n$$

Scalability results for DD stencil computations

- With tree-based (logarithmic) global reductions and scalable nearest neighbor hardware:
  - optimal number of processors scales *linearly* with problem size
- With 3D torus-based global reductions and scalable nearest neighbor hardware:
  - optimal number of processors scales as *three-fourths* power of problem size (almost "scalable")
- With common network bus (heavy contention):
  - optimal number of processors scales as *one-fourth* power of problem size (not "scalable")







Four potential limiters on scalability in large-scale parallel PDE-based codes

- **Insufficient localized concurrency**
- **Load imbalance at synchronization points**
- **Interprocessor message latency**
- **Interprocessor message bandwidth**

"horizontal aspects"

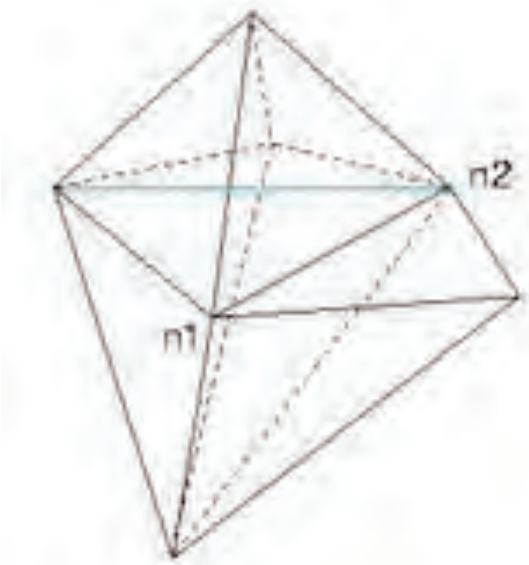
Four potential limiters on arithmetic performance

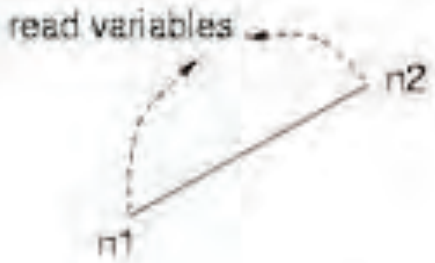
- **Memory latency**
  - failure to predict which data items are needed
- **Memory bandwidth**
  - failure to deliver data at consumption rate of processor
- **Load/store instruction issue rate**
  - failure of processor to issue enough loads/stores per cycle
- **Floating point instruction issue rate**
  - low percentage of floating point operations among all operations

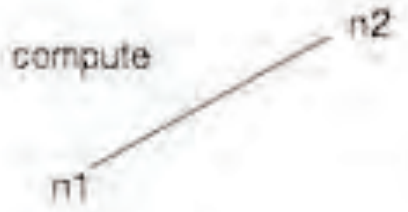
"vertical aspects"

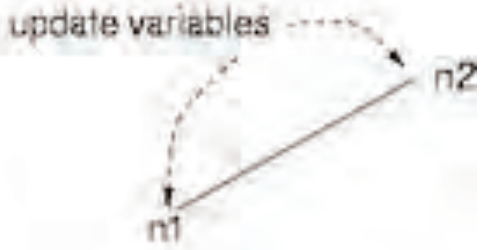
Typical compute-intensive kernel: the edge-based loop

- Vertex-centered grid
- Traverse by edges
  - load vertex values
  - compute intensively
    - e.g., for compressible flows, solve 5x5 eigen-problem for character-istic directions and speeds of each wave
  - store flux contributions at vertices
- Each vertex appears in approximately 15 flux computations









Variables at each node:
density,
momentum (x,y,z),
energy,
pressure

Variables at edge:
identity of nodes,
orientation(x,y,z)
normal area

Candidate stresspoints of PDE kernels

- Vertex-based loops
  - memory bandwidth
- Edge-based “stencil op” loops
  - load/store (register-cache) bandwidth
  - internode bandwidth
- Sparse, narrow-band recurrences
  - memory bandwidth
  - internode bandwidth, internode latency, network diameter
- Inner products and norms
  - memory bandwidth
  - internode latency, network diameter

Summary of observations for CFD case study (aerodynamics simulation – 1999 Bell Prize)

- Processor scalability is *no problem*, in principle
  - if network is richly connected
- For fixed-size problems, global synchronization and near neighbor communication are eventually bottlenecks (strong scaling)
- Coarse grids in hierarchical solvers can become bottlenecks
  - coarse grid concurrency may need to be coarser than fine grid concurrency (recur: multigrid)
- Memory latency is not a serious problem, in principle
  - due to predictability of memory transfers in PDEs
- Memory bandwidth is a *major* bottleneck
- Processor Load-Store functionality *may* be a bottleneck
- Infrequency of floating point instructions in unstructured problems *may* be a bottleneck

Communication optimality for linear solvers

- **Goal: minimize bandwidth and latency costs**
  - **for sequential (memory hierarchies) *or* parallel (networks)**
  - **time_per_flop << 1/BW << latency, and growing apart exponentially**
- **Results:**
  - **new dense and sparse algorithms with less bandwidth and latency than current algorithms (sequential and parallel), by large factors (memory-capacity dependent) with optimality proofs**
  - **performance models predict some large speedups on petascale machines (when communication dominates usual algorithm)**
  - **implementations demonstrate good speedups**

Resource limitations of dwarfs

Dwarf	Performance Limit: Memory Bandwidth, Memory Latency, or Computation?
1. Dense Matrix	Computationally limited
2. Sparse Matrix	Currently 50% computation, 50% memory BW
3. Spectral (FFT)	Memory latency limited
4. N-Body	Computationally limited
5. Structured Grid	Currently more memory bandwidth limited
6. Unstructured Grid	Memory latency limited
7. MapReduce	Problem dependent
8. Combinational Logic	CRC problems BW; crypto problems computationally limited
9. Graph traversal	Memory latency limited
10. Dynamic Programming	Memory latency limited
11. Backtrack and Branch+Bound	?
12. Construct Graphical Models	?
13. Finite State Machine	Nothing helps!

Figure 9. Limits to performance of dwarfs, inspired by an suggestion by IBM that a packaging technology could offer virtually infinite memory bandwidth. While the memory wall limited performance for almost half the dwarfs, memory latency is a bigger problem than memory bandwidth

Recap on scalable algorithms

- We are at a challenging crossroads at the petascale, architecturally, and facing enormous opportunity, scientifically
- We must take up anew algorithmic challenges, including:
  - expose orders of magnitude more concurrency
  - make global synchronization relatively infrequent
  - better exploit multiple precisions
  - make dynamic trade-offs between the storage/transmission of working iterates and their recomputation

  } architec-turally motivated

  - apply subspace projection methods beyond geometric (domain decomposition) to algebraic analogs in multiphysics coupling
  - combine domain-based parallelism efficiently with other "dimensions" – phase-space and energy group variables, stochastic variables, design variables, uncertain parameters

  } applica-tions motivated

- Lots of "virgin territory" for basic algorithm research
  - your dissertation topic could be in the list above