High-Performance Numerical Algorithms for Large-Scale Simulation

Science at Extreme Scales: Where Big Data Meets Large-Scale Computing

Tutorials
Thanks to

- David Keyes, KAUST
- Rob Neely, LLNL
- And others...
Hardware improvements are not enough

Machine improvements tend to improve base or coefficient

Model and algorithm improvements can improve exponent

$O(N^p)$
Definition: Scalability

- The ability of a system or code’s capabilities to increase commensurate with additional resources or cost
  - Hardware scalability typically refers to the cost
    • e.g., All-to-all interconnects between N processors or nodes are fine for small values of N, but are cost prohibitive at large N
  - Algorithmic scalability typically refers to performance or memory usage relative to number of nodes or processors
    • e.g., Code runs twice as fast if you double the number of processors

- Most algorithms/data sizes have scaling limits and performance will not improve indefinitely
Definitions: Strong vs. Weak Scalability

- **Strong Scaling**
  - Overall *problem size* is fixed
  - Goal is to run same size problem faster as resources are increased
  - Perfect scaling means problem runs in 1/P time (compared to serial)

- **Weak Scaling**
  - *Problem size per processor* is fixed
  - Goal is to run larger problem in same amount of time
  - Perfect scaling means a problem P-times larger runs in same time as single processor run

Courtesy: Steve Smith, LLNL
Definition: Granularity

- **The amount of computation performed by a task**
  - Often in relation to frequency of communication

- **Coarse-grained**
  - Lots of work (between communication or sync)
  - Smaller number of infrequent communication tasks
  - MPI: perform a lot of computation before “hitting the network”
  - Communication requirements often reduced by replicating portions of memory from neighboring tasks (“ghost elements”) 

- **Medium-grained**
  - Relatively little work between communication
  - Larger number of smaller tasks/threads
  - Threads (shared-memory accesses) typically incur less overhead

- **Fine-grained**
  - Instruction-level parallelism (e.g. vectors or SIMD)
  - Hardware (and compiler) support to minimize overhead/contention

Modern HPC applications must account for all levels of granularity, and use the corresponding hardware features as appropriate

It’s a balancing act: Finer granularity means more opportunity for parallelism, but a corresponding need for more synchronization (communication)
Definitions: Parallel Speedup and Efficiency

- **Parallel Speedup** is a commonly reported metric
  - Primarily for strong scaling studies
  - In its simplest form is just ratio of time
  - Example:
    - 1 processor run takes 100s
    - 16 processors take 10s
    - 10x speedup

- **Parallel Efficiency**
  - Measures closeness to *ideal* speedup – usually expressed as a percentage
  - Above example: $10 / 16 = 62.5\%$ parallel efficient
  - Also useful for weak scaling studies
    - Replace total time with a time-per-work-unit, e.g., “Grind time” $= \mu s$/zone/cycle

These metrics may or may not be based on a serial (single processor) run; Strong scaling studies are often limited in the dynamic range of processor counts.
Definition: Amdahl’s Law
The Importance of Exposing Parallelism

- Potential speedup is limited by the sequential fraction of your code

\[ S(N) = \frac{N}{P + N(1 - P)} \]

- \( S \) = Theoretical maximum speedup
- \( N \) = number of processors
- \( P \) = fraction of code that can be parallelized

Bottom line: You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!

(99% ➔ 90x, 99.9% ➔ 500x)
Definition: Shared memory vs distributed memory

- **Shared memory**
  - Common address space across all cores
  - Communication done through shared addresses/variables
  - Does not scale well beyond O(75) cores
    - Assuming cache coherence

- **Distributed memory**
  - Address space is local to each node
  - Explicit communication between tasks via network
  - Demonstrated to be highly scalable

- **Current machines are hybrids**
  - Shared memory within a CPU or node, distributed memory between nodes
Definition: Race Conditions

- Errors that occur when two or more processes access the same memory location and one access is for writing
- Often seen in concurrent programming
- Often non-deterministic, i.e., a “Heisenbug”
Well established resource trade-offs

- **Minimize communication**
  - Communicate less often
  - Consolidate messages to hide latency
  - *Communication-avoiding* algorithms

- **Overlap communication**
  - Do useful work while waiting for data
  - *Communication-hiding* algorithms

- **Minimize synchronization**
  - Perform extra flops between global reductions or exchanges to require fewer global operations
  - *Asynchronous* algorithms

- **Bandwidth vs FLOPs**
  - Do more work for every byte transferred
  - *High operational intensity* algorithms
How are most scientific simulations implemented at the petascale today?

- **Iterative methods based on data decomposition and message-passing**
  - Data structures are distributed
  - Each processor works on a subdomain of the original
  - Information exchanged with processors with data with which interactions are required to update
  - Computation and neighbor communication are parallelized, with their ratio constant in weak scaling

- **The programming model is BSP/SPMD/CSP**
  - Bulk Synchronous Programming
  - Single Program, Multiple Data
  - Communicating Sequential Processes

- **Almost all “good” algorithms in linear algebra, differential equations, integral equations, signal analysis, etc., like to globally synchronize – and frequently!**
  - Inner products, norms, pivots, fresh residuals are “addictive” idioms
  - Tends to hurt efficiency beyond 100,000 processors
  - Can be fragile for less concurrency: algorithmic load imbalance, hardware performance variation, etc
Different classes of problems have different characteristics that inherently make concurrency easier (or not)

<table>
<thead>
<tr>
<th>Hyperbolic PDEs</th>
<th>Parabolic PDEs</th>
<th>Elliptic PDEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\partial_t u + a \partial_x u = 0$</td>
<td>$\partial_t T = \mu \nabla^2 T$</td>
<td>$\nabla^2 \phi = \rho$</td>
</tr>
<tr>
<td>▪ Advection and wave propagation</td>
<td>▪ Diffusion evolution: “To slump”</td>
<td>▪ Equilibrium problem</td>
</tr>
<tr>
<td>▪ No dissipation</td>
<td>▪ Infinite wave speeds</td>
<td>▪ Steady-state</td>
</tr>
<tr>
<td>▪ Finite wave speeds</td>
<td>▪ Implicit time stepping</td>
<td>▪ Global dependence</td>
</tr>
<tr>
<td>▪ Explicit time stepping</td>
<td>▪ Global dependence</td>
<td></td>
</tr>
<tr>
<td>▪ Local dependence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Real problems exhibit combinations of these behaviors
What types of problems occupy major supercomputer centers?

- **Linear algebra on dense symmetric/Hermitian matrices**
  - Hamiltonians (Schrödinger) in chemistry/materials
  - Hessians in optimization
  - Schur complements in linear elasticity, Stokes, and saddle points
  - Covariance matrices in statistics

- **Poisson solves**
  - Highest order operator in many PDEs in fluid and solid mechanics, EM, DFT, MD, etc.
  - Diffusion, gravitation, electrostatics, incompressibility, equilibrium, Helmholtz, image processing – even analysis of graphs
Krylov Subspace Methods

- Iterative methods for solving large-scale linear systems
- "Matrix free" - Only require action of matrix on a vector
  \[ \mathcal{K}_n(A, y) = \text{span} \left( y, Ay, \ldots, A^{n-1}y \right) \]
- Search for an approximate solution to \( Ax = b \) in the subspace
  \[ \mathcal{K}_n(A, r_0) \quad r_0 = b - Ax_0 \]
- Examples:
  - Conjugate Gradient (CG) [Symmetric, positive-definite systems (SPD)]
  - Generalized Minimum Residual (GMRES) [Nonsymmetric systems]
  - Biconjugate Gradient (BiCGSTAB) [Nonsymmetric systems]

Domain decomposition choices

- Partitioning of a domain is often done by using a graph representation
  - Element based (FEM)
  - Edge-based
  - Vertex-based

- Domain Decomposition methods are characterized by four decisions
  - Type of partitioning
  - Overlap
  - Processing of interface values
  - Subdomain solution method
As part of a divide-and-conquer approach, we partition a domain into subdomains.

The linear system has the general block form

\[
\begin{pmatrix}
B_1 & & \\
& E_1 & \\
& & E_2 \\
& & & \ddots \\
& & & & E_s \\
F_1 & & & & B_s \\
F_2 & & & & E_s \\
\vdots & & & & \ddots \\
F_s & & & & C \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
\vdots \\
x_s \\
y \\
\end{pmatrix}
= 
\begin{pmatrix}
f_1 \\
f_2 \\
\vdots \\
f_s \\
g \\
\end{pmatrix}
\]

or

\[
\begin{pmatrix}
B & E \\
F & C \\
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
\end{pmatrix}
= 
\begin{pmatrix}
f \\
g \\
\end{pmatrix}
\]

where \( x \) are interior unknowns and \( y \) are interface unknowns.
Solving first for $x$ in the system

$$\begin{pmatrix} B & E \\ F & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}$$

one can write the reduced system

$$Sy = (C - FB^{-1}E)y = g - FB^{-1}f$$

where $S$ is the Schur Complement

If this system can be solved, all of the interface variables will be known

— All of the subdomains then decouple and can be solved in parallel

— Global Schur complement can often be assembled from local Schur complements
Schwarz Alternating Methods

- **Multiplicative**
  - Solve on each subdomain independently
  - Use lagged interface data from other domain(s)
  - Iterate to convergence
  - Related to Block Gauss Seidel on the local Schur complement system

- **Additive**
  - Like multiplicative, but components in each subdomain are only updated after a complete cycle across the whole domain completes
  - Similar to Block Jacobi iteration
Parallel Preconditioning Strategies

- **Block-Jacobi**
  - Use an additive projection onto a specific set of subspaces
  - Overlap regions are weighted and added

- **Polynomial Preconditioners**
  - Preconditioner is defined as low-degree polynomial in $A$
  - Can be constructed using only matvecs
  - Chebyshev Acceleration
    - Optimal in the sense that preconditioned matrix is close to identity
    - No inner products

- **Multicoloring**
  - Graph coloring techniques
    - Adjacent nodes have different colors
  - Nodes of same color determined simultaneously in ILU sweeps
  - Red-Black (Re-)Ordering
    - Block diagonal matrices are made diagonal
    - Highly parallel solution
Multigrid (MG) uses a hierarchical sequence of coarse grids to accelerate the fine grid solution.

- **smoothing** (relaxation)
- **restriction**
- **prolongation** (interpolation)

Multigrid solvers have $O(N)$ complexity and hence have good scaling potential.
Algebraic Multigrid (AMG) is based on MG principles, but only uses matrix coefficients

- Unstructured grids lack simple coarsening rules
- Automatically coarsens “grids”
- Error left by pointwise relaxation is called algebraically smooth error
  - Not always geometrically smooth
- Weak approximation property: interpolation must interpolate small eigenmodes well

\[ \|E_{TG}\|_A^2 \leq 1 - \frac{1}{K}; \quad K = \sup_e \|A\| \frac{\| (I - P[0\ I])e \|_2}{\|e\|_A^2} \]

- Near null-space is important!
AMG grid hierarchies for several 2D problems

domain1 - 30°

domain2 - 30°
pile
square-hole
Straightforward MG parallelization yields optimal-order performance for V-cycles

- ~ 1.5 million idle cores on Sequoia, but still performs optimally

- 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
  $$T_V = O(\log N)(\text{comm latency}) + O(\Gamma_p)(\text{comm rate}) + O(\Omega_p)(\text{flop rate})$$
Parallel AMG 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
Realistic models of materials and biomolecules require very large ab initio simulations

**Standard algorithms**
- Efficient only up to 500 atoms
- Reaching limits on today’s largest computers
- Have $O(N^3)$ complexity and global communications

**New $O(N)$ algorithm with short-range communications only**
- Represent electronic structure as set of localized functions (cf. eigenfunctions)
- Use approximate inverse strategy to calculate coupling between these functions (compute selected elements of Gram matrix inverse)

**Controllable accuracy with $O(N)$ approximations**

**Demonstrated excellent weak scaling up to 100,000 atoms**

New algorithm allows fast and accurate solutions in $O(N)$ operations for 100K atoms

Courtesy of Jean-Luc Fattebert, ORNL
Taskification based on Directed Acyclic Graphs (DAGs)

- **Advantages**
  - Remove artifactual synchronizations in the form of subroutine boundaries
  - Remove artifactual orderings in the form of pre-scheduled loops
  - Expose more concurrency

- **Disadvantages**
  - Pay overhead of managing task graph
  - Potentially lose some memory locality
Loop nests and subroutine calls, with their over-orderings, can be replaced with DAGs

- Diagram shows a dataflow ordering of the steps of a 4×4 symmetric generalized eigensolver
- Nodes are tasks, color-coded by type, and edges are data dependencies
- Time is vertically downward
- Wide is good; short is good
Loops can be overlapped in time

Green, blue and magenta symbols represent tasks in separate loop bodies with dependences from an adaptive optics computation.

Tasks from 3 loops of optical “reconstructor” pipeline are executed together.

c/o H. Ltaief (KAUST) & D. Gratadour (OdP)
Hierarchically low-rank operators

- **Advantages**
  - Shrink memory footprints to live higher on the memory hierarchy
    - Higher means quick access
  - Reduce operation counts
  - Tune work to accuracy requirements
    - e.g., preconditioner versus solver

- **Disadvantages**
  - Cost of compression
  - Not all operators compress well
Key tool: Hierarchical matrices

- [Hackbusch, 1999]: off-diagonal blocks of typical differential and integral operators have low effective rank

- By exploiting low rank, \( k \), memory requirements and operation counts approach optimal in matrix dimension \( n \):
  - From \( O(n^2) \) to \( O(k n \log(n)) \)
  - Constants carry the day

- Such hierarchical representations navigate a compromise
  - Fewer blocks of larger rank ("weak admissibility")
  - More blocks of smaller rank ("strong admissibility")
Example: 1D Laplacian

\[
A = \begin{bmatrix}
  2 & -1 & & & \\
-1 & 2 & -1 & & \\
& -1 & 2 & -1 & \\
& & -1 & 2 & -1 \\
& & & -1 & 2 \\
\end{bmatrix}
\]

\[
A^{-1} = \frac{1}{8} \times \begin{bmatrix}
  7 & 6 & 5 & & & \\
 6 & 12 & 10 & & & \\
 5 & 10 & 15 & & & \\
 4 & 8 & 12 & 16 & 12 & 8 & 4 \\
 3 & 6 & 9 & 12 & 15 & 10 & 5 \\
 2 & 4 & 6 & 8 & 10 & 12 & 6 \\
 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  0 & 0 \\
  0 & 1 \\
\end{bmatrix} \begin{bmatrix}
  -1 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  1 \\
  2 \\
  3 \\
\end{bmatrix} \begin{bmatrix}
  4 & 3 & 2 & 1 \\
\end{bmatrix}
\]
“Standard (strong)” vs. “weak” admissibility

After Hackbusch, et al., 2003
Block Structured Adaptive Mesh Refinement provides another hierarchical approach to focusing effort where it is most needed

- Domain is decomposed into disjoint rectangular patches
- Solution is updated from coarse to fine patches
- Corrections are propagated from fine to coarse patches
- Fine patches are subcycled time accurately
- Global composite solve needed for parabolic/elliptic problems
While hardware improvements have provided gains, algorithmic improvements also play a big role in achieving high performance.

The best algorithms have common features:
- Hierarchical structures that minimize communication
- Divide-and-conquer
- Large amounts of local work that minimize impact of communication
- Avoidance of unnecessary global operations (norms, inner products, etc)