Is There A Moore’s Law For Algorithms?

David E. Womble

Sandia National Laboratories

Presented at Salishan

April 19, 2004

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of energy under Contract DE-AC04-94AL85000

So, is there a Moore's Law for algorithms

Sorry. NO.

(We've done better than that.)

So What Are Algorithms?

- **“A procedure for solving a mathematical problem in a finite number of steps that frequently involves the repetition of an operation; a step-by-step procedure for solving a problem or accomplishing some end, especially by computer.”**

  - Webster’s

- **“Any mechanical or recursive procedure”**

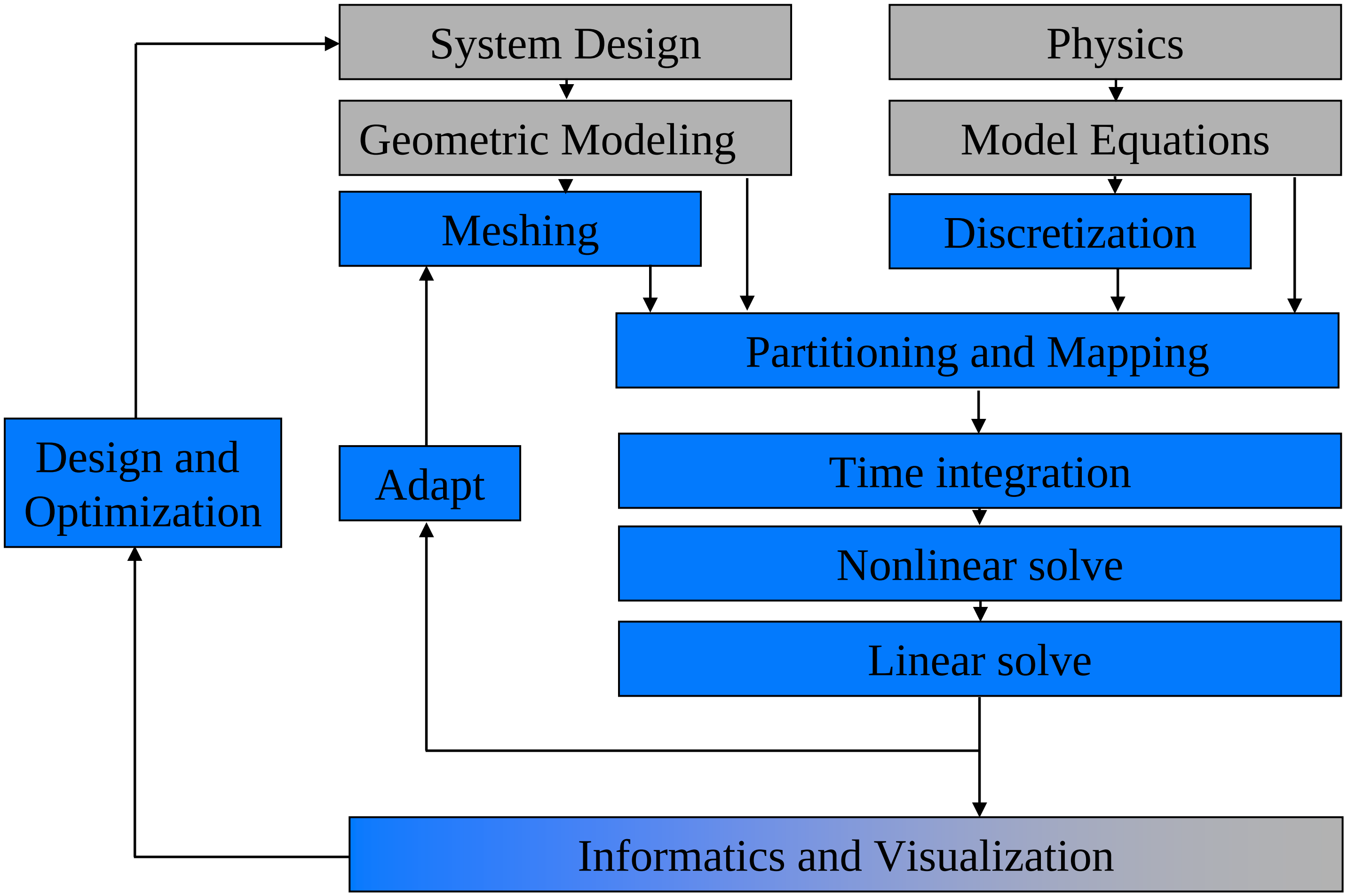
  - American Heritage Dictionary

- **How important are numerical/computational algorithms?**

Have you ever tried to do long division with Roman numerals?

Sandia National Laboratories

Restricting the Scope
(Model-based, Scientific Computing)



Where Were HW and SW in 1980?

- Hardware
  - Big Iron
    - CDC 7600
    - Cray 1
  - Microprocessors
    - 6800
    - Z80
    - 8086
  - Departmental Computing
    - Vax 11/780
  - Business
    - DEC 20
    - IBM 370

- Software
  - Scientific Computing
    - Fortran
    - Assembly
  - Business
    - COBOL
  - Research
    - C
    - APL
    - Lisp
  - Casual
    - Basic

Where Were The Algorithms?

- Finite elements were becoming popular
  - Becker, Carey and Oden was almost done
- Production algorithms
  - RK4
  - Newton/Newton-Raphson
  - Banded/Sparse LU
  - Jacobi/Gauss-Seidel
- SLATEC was the DOE numerical library of choice
- Research algorithms
  - Preconditioned Conjugate Gradient (PCG)
  - Multigrid
  - Golub and van Loan was not out.
- Minimal optimization or system studies, although there was a fair amount of research being done on basic algorithms

Putting It Together, What Could We Do?

- Large calculations
  - 2D
  - Finite differences
  - Semi-implicit calculations
  - 3000 variables in an implicit solver
  - 100,000 in an explicit solver

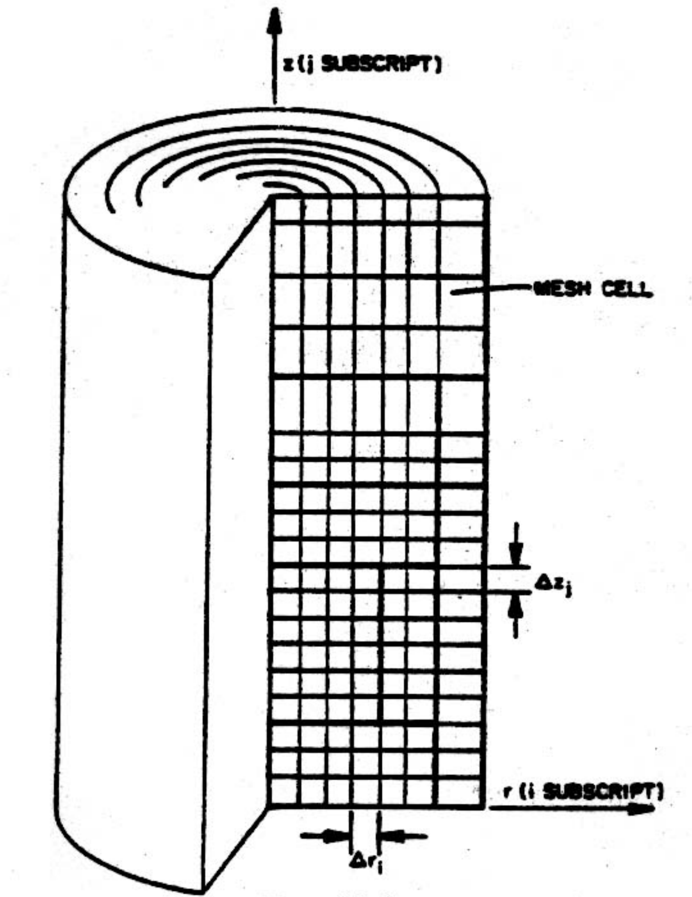
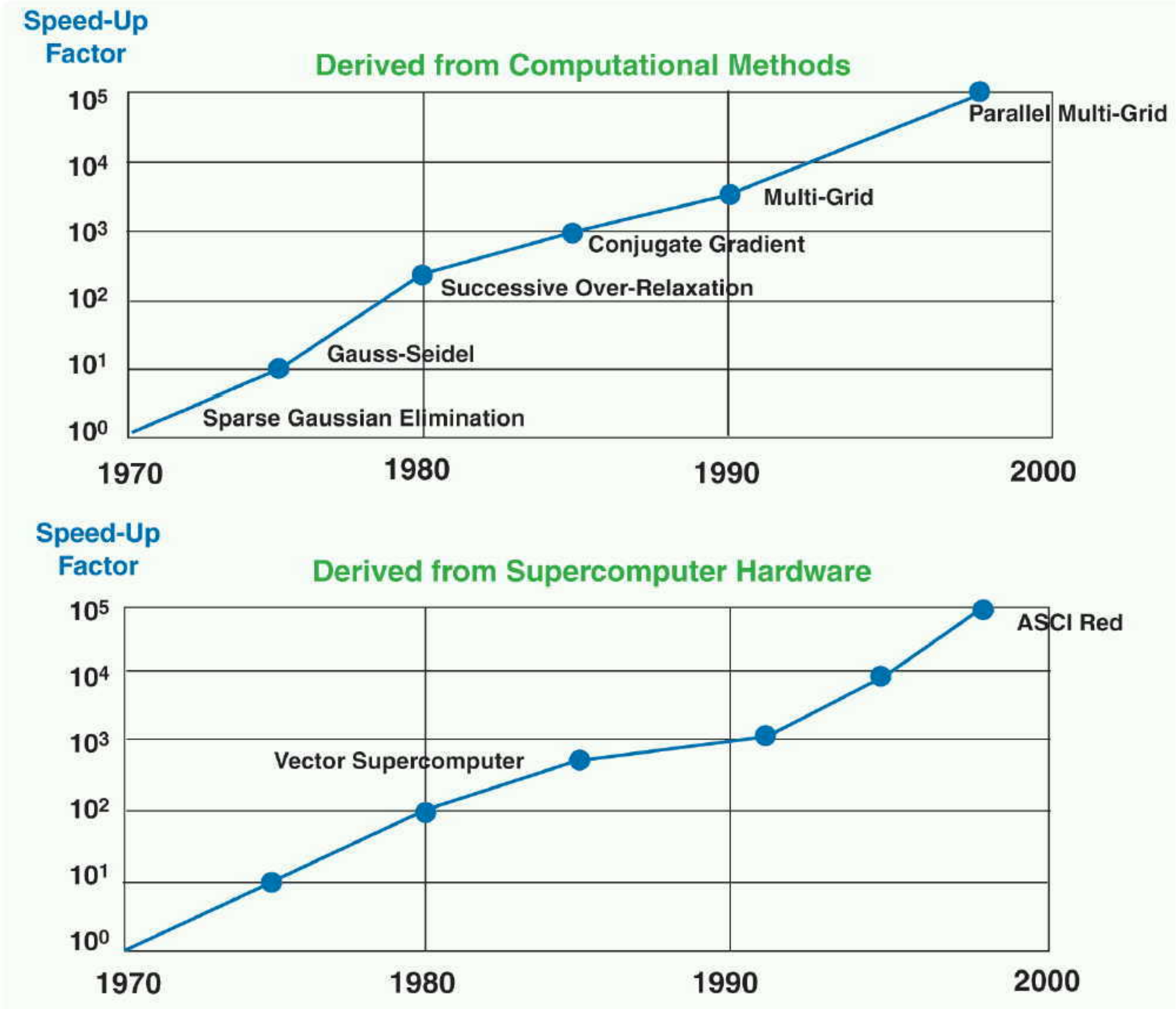


Fig. IV-2.

What Drives Algorithms?

- Algorithmic complexity
  - A bad serial algorithm doesn’t make a good parallel algorithm.
- Parallel complexity
- Memory architecture
- Parallel implementation

A Case Study - Solvers

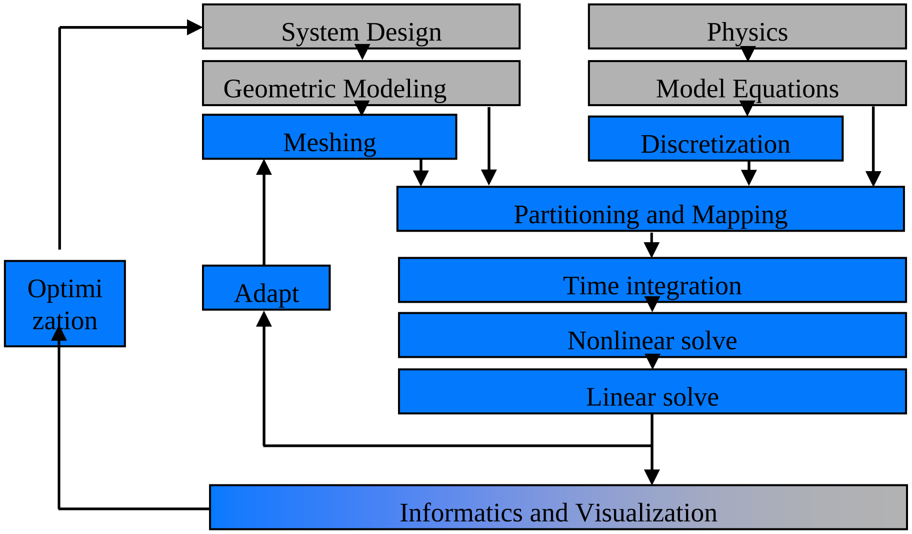


From SIAM Review, 2001

Why Look At Solvers?

- A lot of algorithms work (in scientific computing) has been focused on solvers because they are the computational kernel of many codes.
  - Linear solvers
  - Nonlinear solvers
  - Time integrators

- Classical linear solvers
  - LU factorization
  - Iterative solvers (Jacobi, GS, Steepest descent)
  - Conjugate gradient (CG)
  - PCG
  - MG



- Specialized solvers
  - FFT
  - Multipole
  - Wavelet

- They illustrate the importance of the serial performance.

LU Factorization

- Closely related to Gaussian elimination

- Led the study of numerical algorithms
  - Propagation of errors
  - Stability
  - Full and partial pivoting

- Started as the solver of choice for scientific computing

- Requires O(n^3) flops
  - Factoring a 1000 x 1000 matrix requires 667 million flops.

- In 1980, this could be the inner loop in a nonlinear solver or time-dependent calculation.

- Can be formulated so that it is not bandwidth limited (even though they didn't need to think in those terms until the 1970s).

Banded LU factorization

- It didn’t take long to realize that there were a lot of zeros in a finite element calculation.
- Banded LU requires $O(nk^2)$ flops (k is the bandwidth).
- There were still a lot of reasons to use an explicit time step, even with severe time-step restrictions.
- Now our 667×10^6 flops becomes 667×10^4 flops (for n=1000). That is a factor of 100 improvement, even on a small problem. This is significant, but we can’t increase the size of our simulations by very much.
- Sparse methods also under development.

Conjugate Gradient

- First developed as a direct method in the 1950s.
- The basic idea is that we can minimize an error over an increasing large subspace. When the subspace includes the range of the matrix, the problem is solved.
- This was a nice idea, but it required the same number of flops as LU or sparse LU algorithms.
- CG as a direct method never found its way into production.
- And there we stayed.

Iterative Methods

- **Meanwhile, back at research ranch …**

- **Researchers were working on iterative algorithms.**
  - **Jacobi iteration, steepest descent and other splitting methods were well known.**
  - **Often O(n^3), although relaxation parameters improved that somewhat. That analysis was started in the 1950s.**

- **A key insight was that CG can be treated as an iterative method and preconditioned with other iterative methods to speed up the convergence.**

- **CG preconditioned with basic iterative methods often shows algorithmic scaling between O($n^{1.5}$) and O(n^2).**

- **Suddenly we had another factor of 100 for our small problem and for a large problem, we get factors of 1000s.**

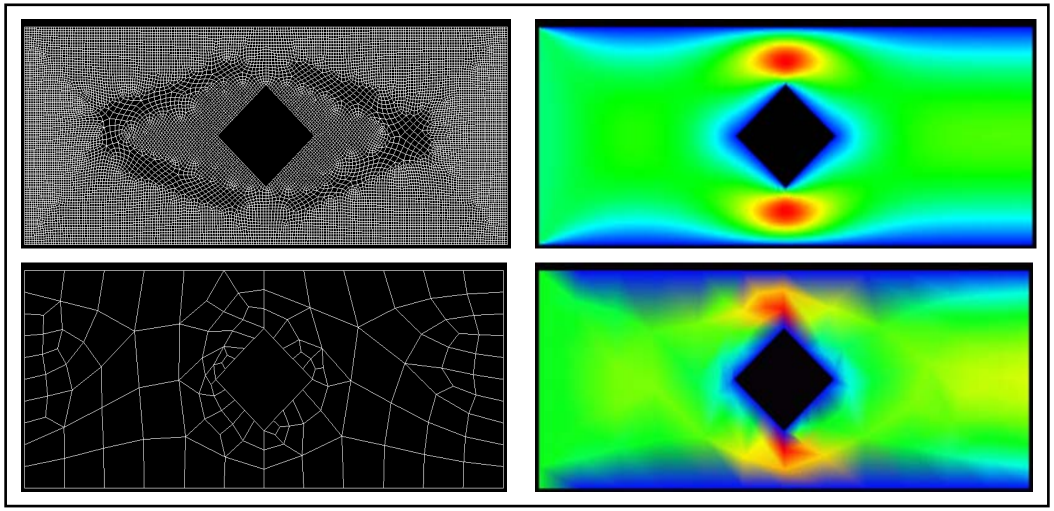
Multigrid

- But the algorithmic scaling was still sub-optimal. Iterative methods tended to damp out some frequencies quickly and others veeeerrrrrrrrrrry slowly.

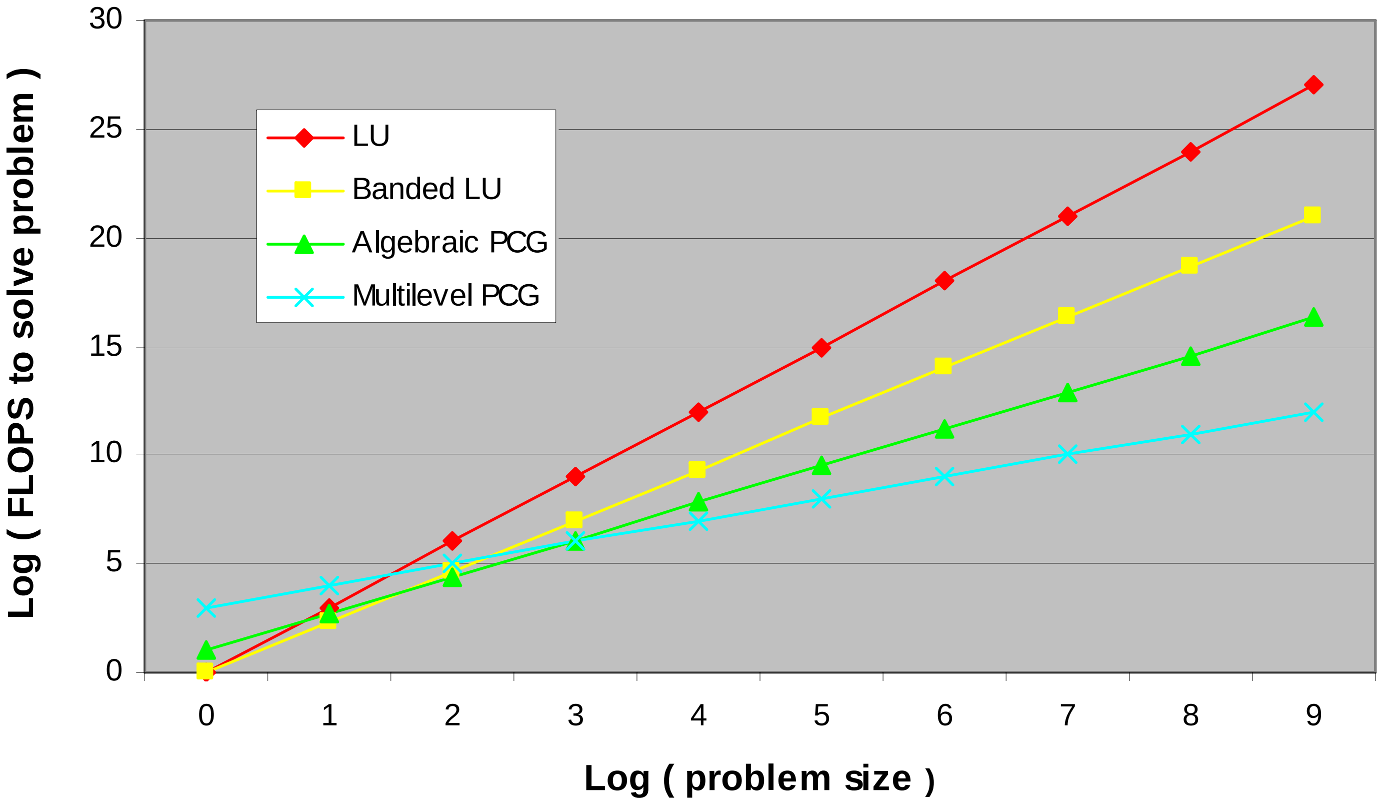
- One answer is multigrid which was developed in the early 1970s although the ideas were around before that.

- MG has optimal algorithmic scaling of O(n).

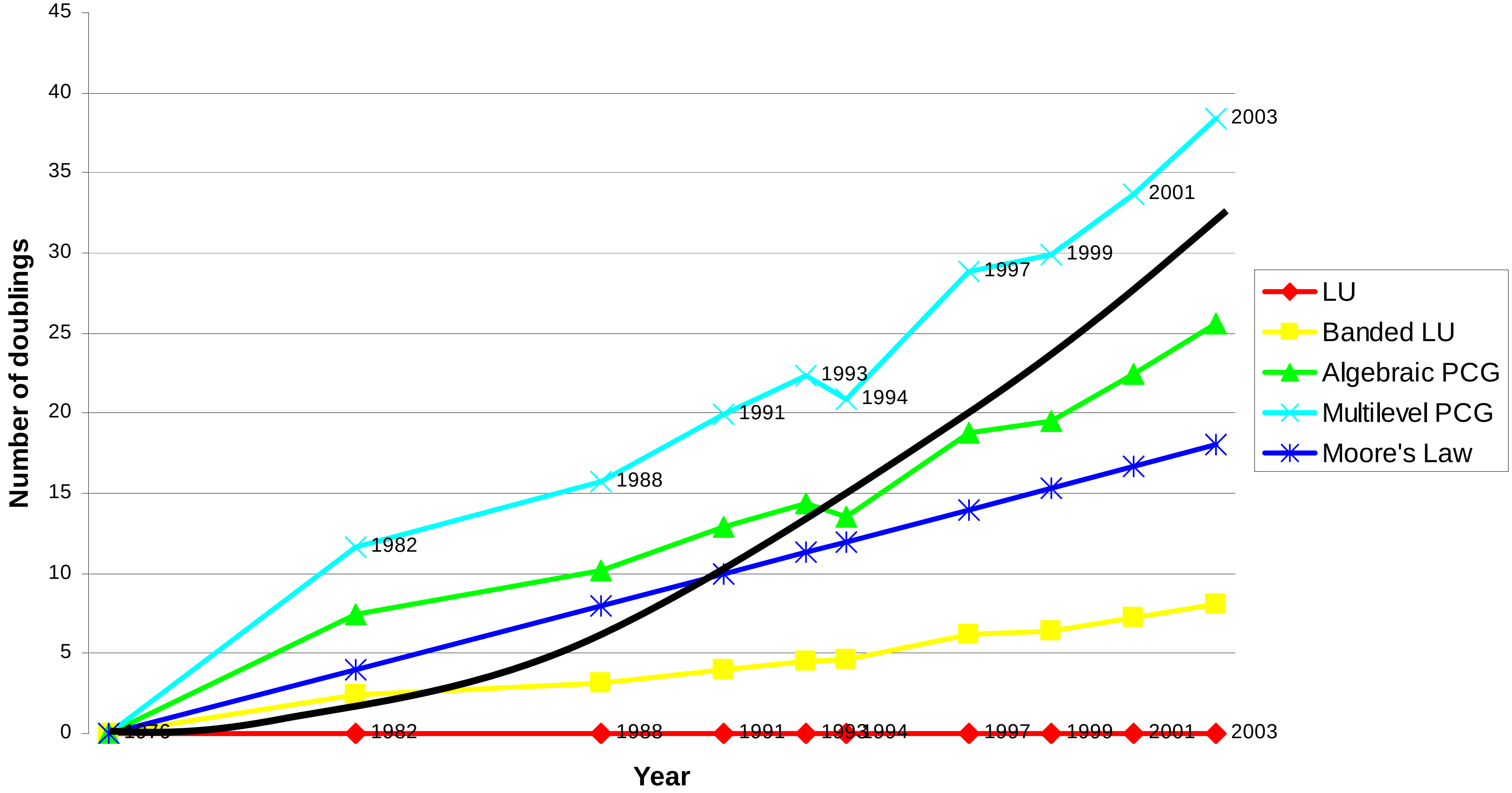
- But nothing is free, and MG suffers by being very problem dependent and difficult for unstructured grids.



Solver Scaling



The Solver Law



What Drives Algorithms?

- Algorithmic complexity
  - A bad serial algorithm doesn’t make a good parallel algorithm.
- Parallel complexity
- Memory architecture
- Parallel implementation

Architecture -> Algorithms

- What architectures have driven algorithms since then?

- Vectors
  - Also seen as fine-grain parallelism

- Cache (deep memory hierarchies)
  - As processing speed became more affordable than memory bandwidth

- Parallelism

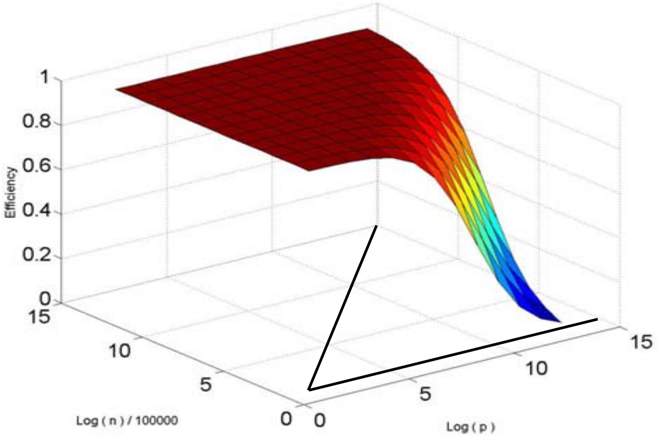
- Parallelism can be viewed as another level in the memory hierarchy, but they are handled differently.

Architecture -> Algorithms 2

- Parallel
  - The idea was not new
    - "When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes." L. F. Menabrea, 1842, “Sketch of the Analytical Engine invented by Charles Babbage”
  - MPP became inevitable between 1986 and 1987.
    - High levels of integration
    - Amdahl’s Law (1967) no longer limited our thinking

How Do We Handle The Parallelism?

- **Think logarithmically (e.g., communication patterns)**
- **Pay attention to data distribution (minimize communications)**
- **Don’t send small messages**
- **Use algorithms that are local in nature**
- **Think about how to use the machine**



  - **Fixed-size speedup   T(n,1) / T(n,p)**
  - **Scaled speedup   p*T(n,1) / T(p*n,p)**

    =   T(p*n,1) / T(p*n,p)

    (If algorithmic efficiency perfect)

How Do We Handle the Memory Hierarchy?

- Restructure the codes so that we get as much reuse as possible
  - In LU factorization, this means working on matrices instead of vectors and using level 3 BLAS.
  - In finite element codes, this means doing as much work on one element before moving on to the next
- Some optimizations can be done automatically (ATLAS), and there is a strong dependency on the compiler as well as the hardware.
- But ultimately the answer is recursion.

System Software -> Algorithms

- Have evolved to
  - Fortran
  - C/C++

- System software hasn’t really driven algorithms. In part, this is because there hasn’t been a fundamental change in system software or languages (within the scope of this presentation).

- Instead, compilers and languages and systems have driven how libraries are structured, how much work it takes to deliver an algorithm, how portable implementations are, etc.

Informal SNL/CIS Survey Results

- 7 FFT
- 4 CG
- 3 MG and MG preconditioning
- 2 Simplex method
- 2 Quicksort
- 2 FE methods
- 2 Contact algorithms
- 2 Interior point methods (e.g. Karmaker)
- 2 Metropolis MC sampling
- 1 Multipole
- 1 Mesh-free methods
- 1 Sparse, direct methods
- 1 GMRES
- 1 High-order FV
- 1 Broyden’s method
- 1 Blossom algorithm (graph matching)
- 1 Smith-Waterman (pattern matching)
- 1 Bowyer-Watson (Delaunay triangulation)
- 1 Gear’s method
- 1 Verlet's Leapfrog Molecular Dynamics method
- 1 Genetic algorithms
- 1 Interface capturing
- 1 Cellular automata
- 1 Hashing
- 1 Marching cubes isosurfacing
- 1 Lanczos diagonalization
- 1 Huffman encoding
- 1 RSA

Algorithms Top 10

(Francis Sullivan and Jack Dongarra in *Computing in Science & Engineering*)

- 1946: The Metropolis Algorithm for Monte Carlo. (Through the use of random processes, this algorithm offers an efficient way to stumble toward answers to problems that are too complicated to solve exactly.)
- 1947: Simplex Method for Linear Programming. (An elegant solution to a common problem in planning and decision-making.)
- 1950: Krylov Subspace Iteration Method. (A technique for rapidly solving the linear equations that abound in scientific computation.)
- 1951: The Decompositional Approach to Matrix Computations. (A suite of techniques for numerical linear algebra.)
- 1957: The Fortran Optimizing Compiler. (Turns high-level code into efficient computer-readable code.)
- 1959: QR Algorithm for Computing Eigenvalues. (Another crucial matrix operation made swift and practical.)
- 1962: Quicksort Algorithms for Sorting. (For the efficient handling of large databases.)
- 1965: Fast Fourier Transform. (Perhaps the most ubiquitous algorithm in use today, it breaks down waveforms (like sound) into periodic components.)
- 1977: Integer Relation Detection. (A fast method for spotting simple equations satisfied by collections of seemingly unrelated numbers.)
- 1987: Fast Multipole Method. (A breakthrough in dealing with the complexity of n-body calculations, applied in problems ranging from celestial mechanics to protein folding.)

What’s Next

- Linear solvers
  - We won’t go beyond O(n)
  - Improved “constants” and preconditioners
  - Improved applicability
- Better nonlinear solvers and time integrators
  - Particularly better optimization
- Better numerics, e.g., finite elements and adaptivity
- Improved robustness
- Better enabling technologies
  - Load balancing
  - Meshing
  - Visualization
- The ability to couple length and time scales
- Informatics. (I think the algorithms picture will be very different here.)

Conclusions

- Algorithms have shown Moore’s Law type improvements
- Algorithm improvements have led to significant increases in our modeling and simulation capabilities
- Algorithms will continue to advance, in general and in terms of the problem to which they can be applied.
- There will be big advances in “new” areas such as optimization and informatics
- Need to keep fundamental research going. We haven’t reached the point where we can just pull something off the shelf.

Sandia National Laboratories