Concepts and Algorithms for Computational Science and Engineering

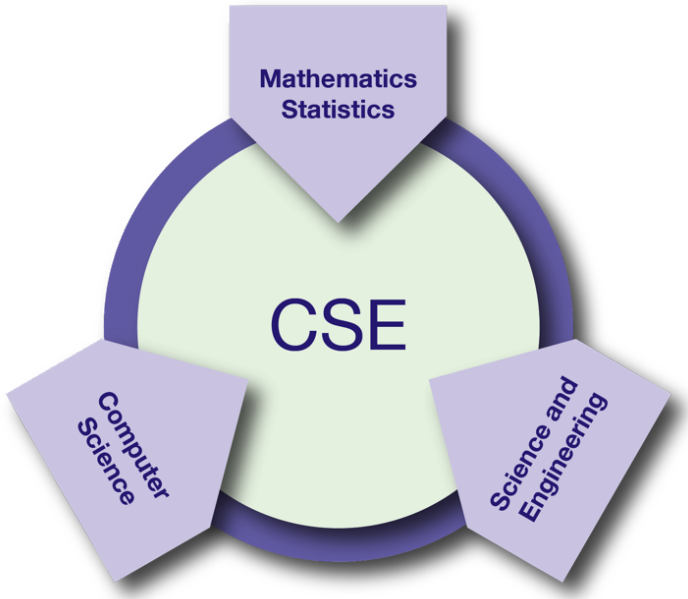
High Performance Computing, PDC Summer School 2018

Michael Hanke

14 August 2018

Outline

Computational Science and Engineering



"CSE is devoted to the development and use of computational methods for scientific discovery in all branches of the sciences and for the advancement of innovation in engineering and technology, and for support of decision-making across a spectrum of societally important application areas. It is a broad and vitally important field encompassing methods of HPC and playing a central role in the data revolution."

Research and Education in Computational Science and Engineering, SIAM Rev 60(2018), 707–754

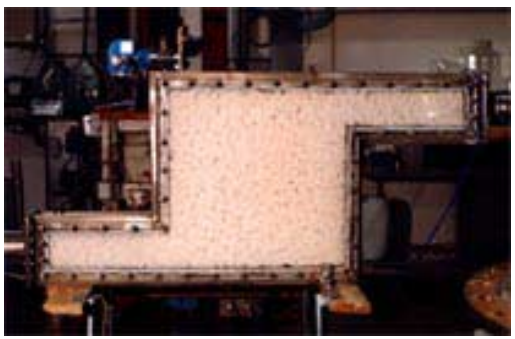
Numerical Analysis

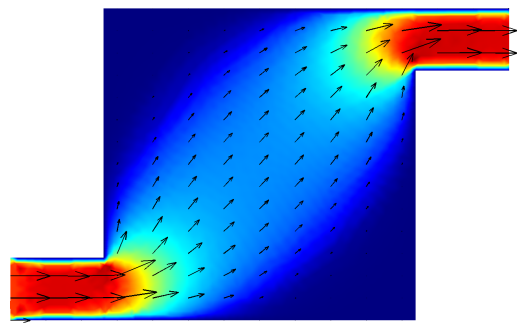
- Original term since the early 1950s
- Today it denotes
  - Developemnt of numerical algorithms from given mathematical models
  - Analysis of these algorithms: consistency, stability, convergence, order of accuracy, computational complexity (Example: error estimate $|u(x_j) - u_j| \leq Ch^p, j = 1, 2, \ldots, J$)

Virtual Prototyping

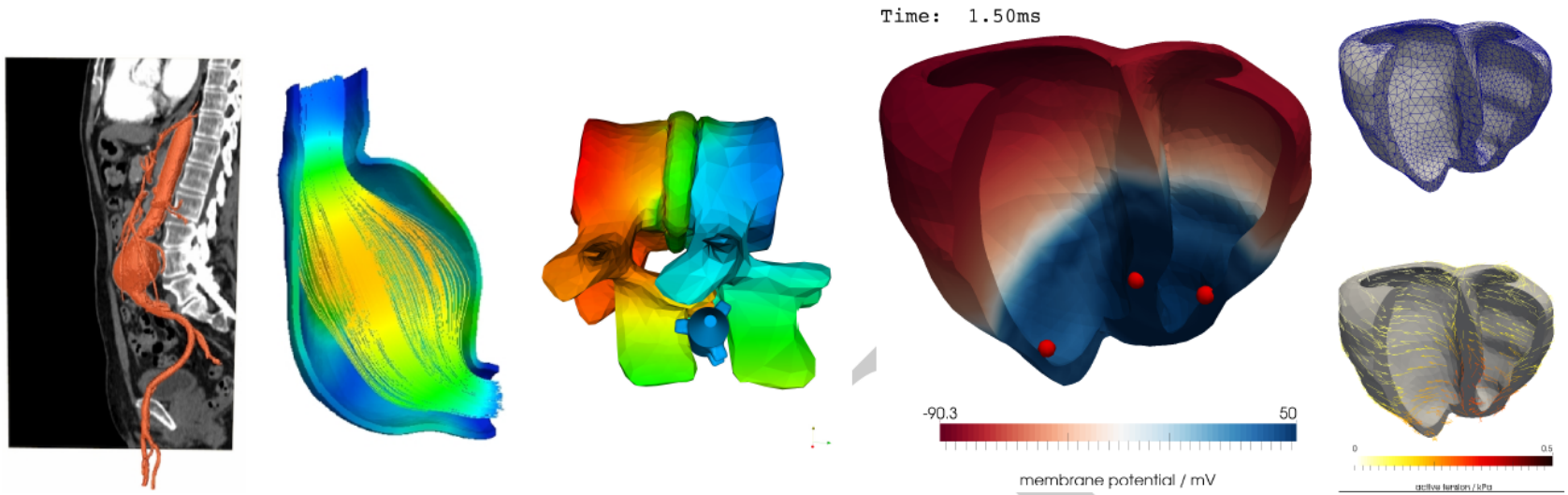
The term virtual prototyping is often used in industry to describe the use of simulation, data base techniques and visualization for

- Understanding
- Verification
- Planning, optimization and control.



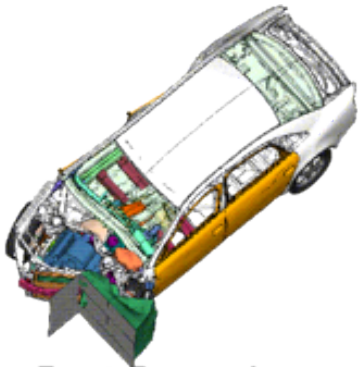


CSE: Medicine

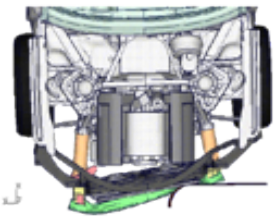


Design and optimization of implants (CFD, FSI); Electrical activation of the human heart (courtesy Rolf Krause)

CSE: Automotive Industry

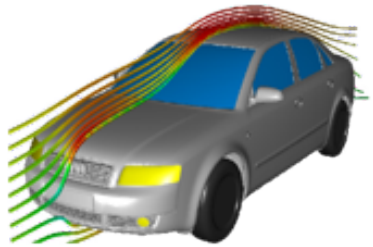


Front, Rear and Side Crash



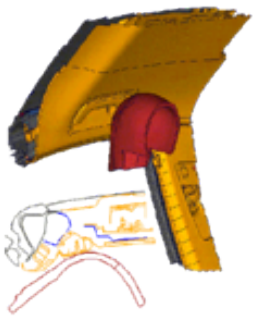
Insurance Test

Pedestrian Protection



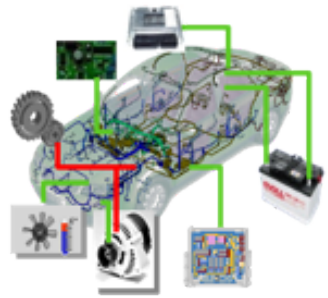
CFD Aerodynamics

Occupant Safety

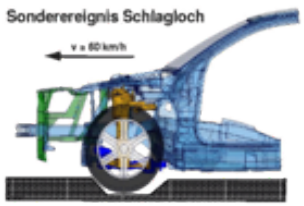


Head Impact

NVH Analyses / Acoustics



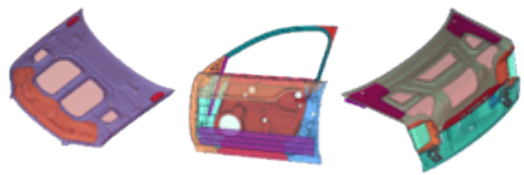
PowerNet



Durability and Fatigue

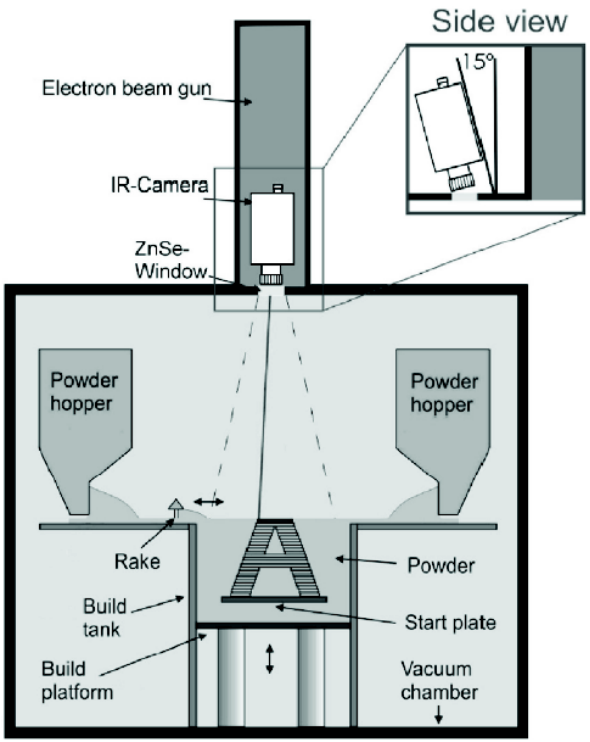
Seats

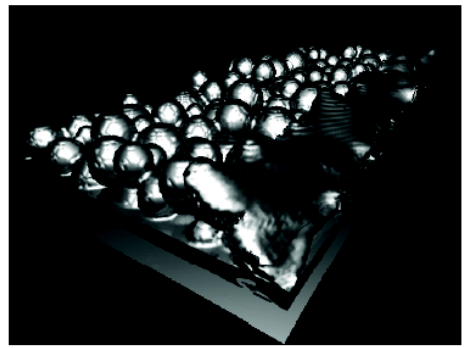
Global and Local Dynamic Stiffnesses

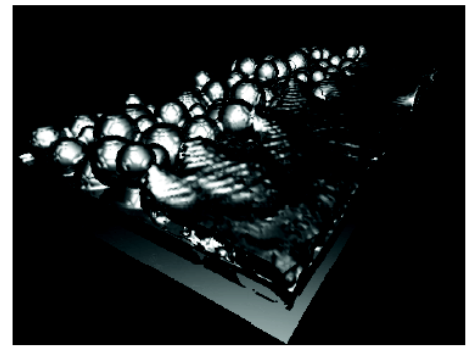


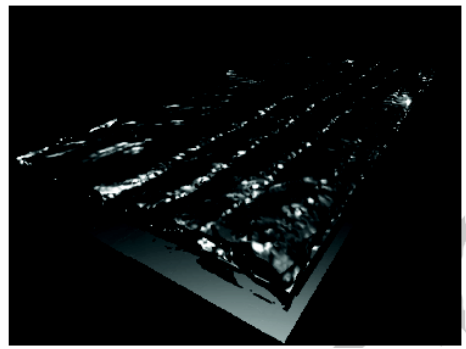
Functional Dimensioning of Doors, Flaps and Hoods

CSE: 3D Printing Process







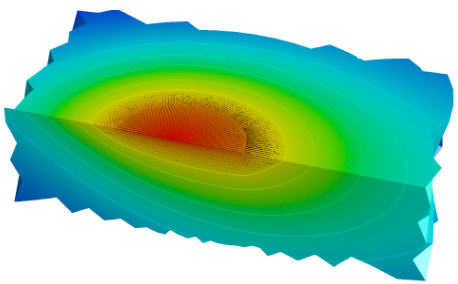


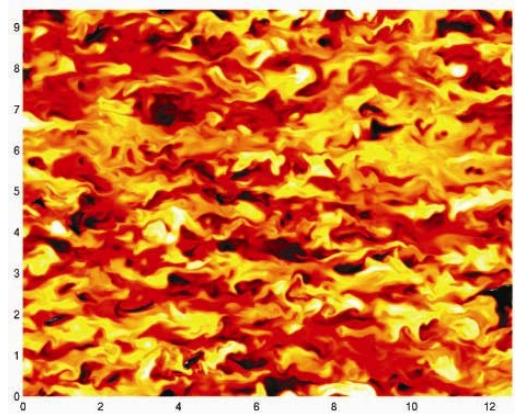
CSE: What's to Come

- Examples from life sciences (molecular biology), material science (phase transition in welding), fluid dynamics (turbulence)
- Level of peta-scale computing: break trough for CSE
- We are on the road to exa-scale computations

What's to Come (cont)

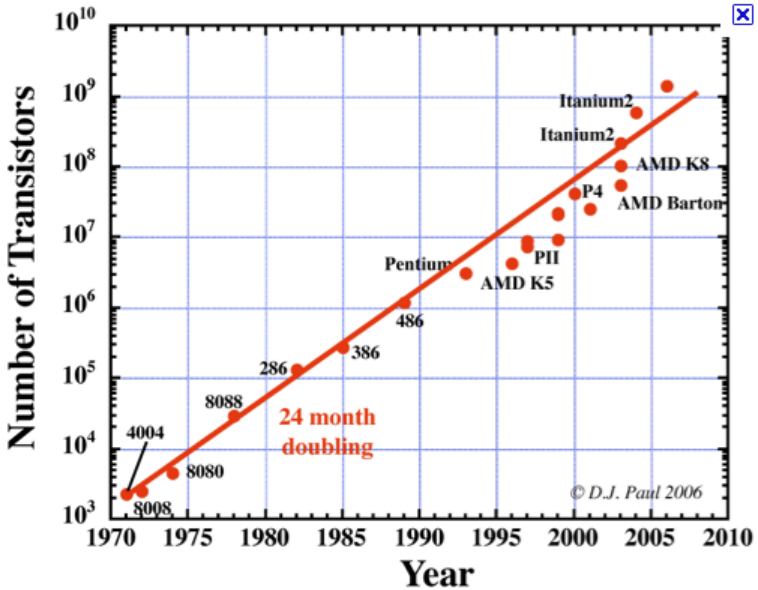
- Ubiquitous parallelism
  - Contention in clock speed and architectural enhancements
  - Pervasive computing (imbedded systems)
- CSE and Data Science: big data analysis
  - "Big data": Often statistical methods

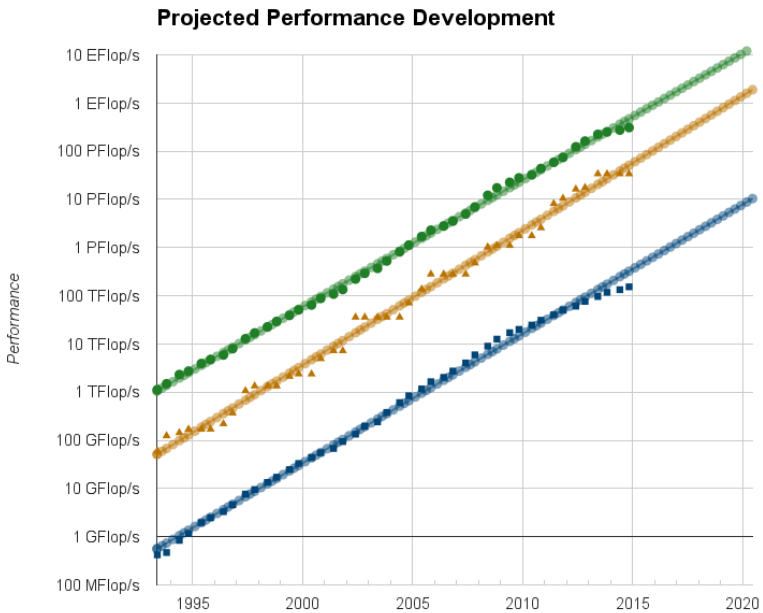




Evolution of Scientific Computing

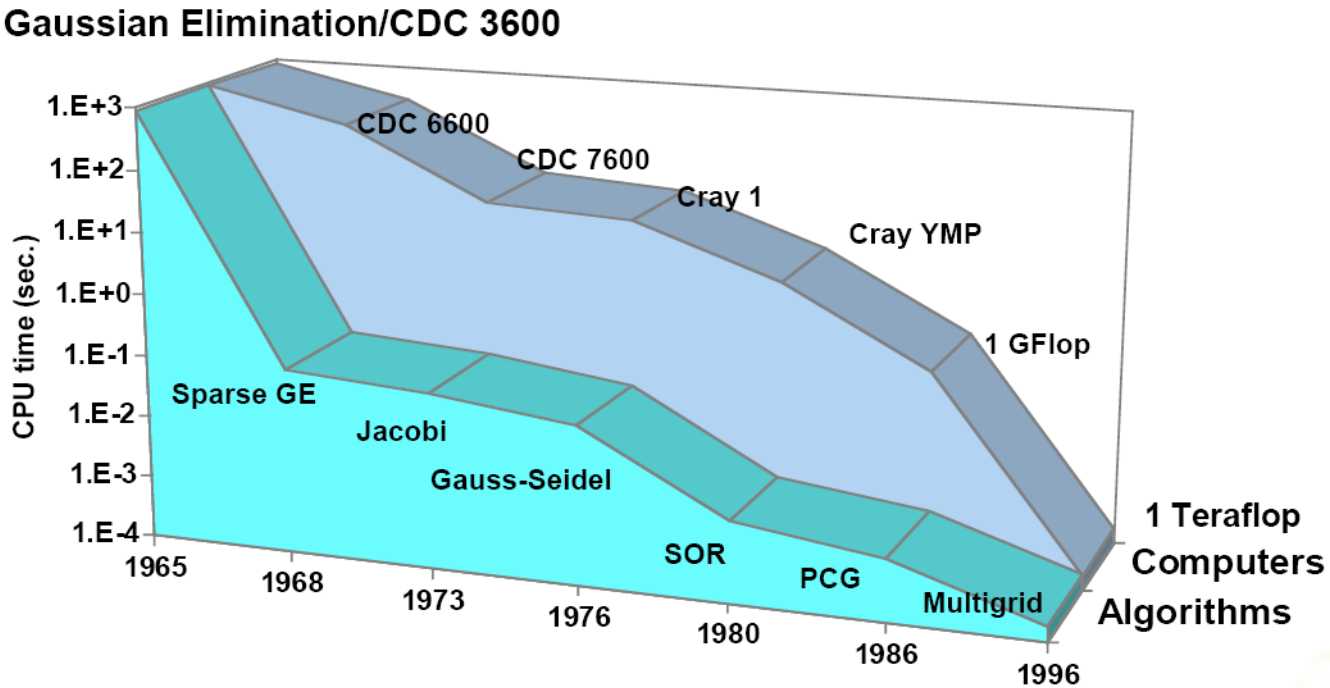
The development of scientific computing is based on progress in computer technology: *Moore's law*





Evolution of Scientific Computing (cont)

The development of scientific computing is also based on progress in algorithms and software.



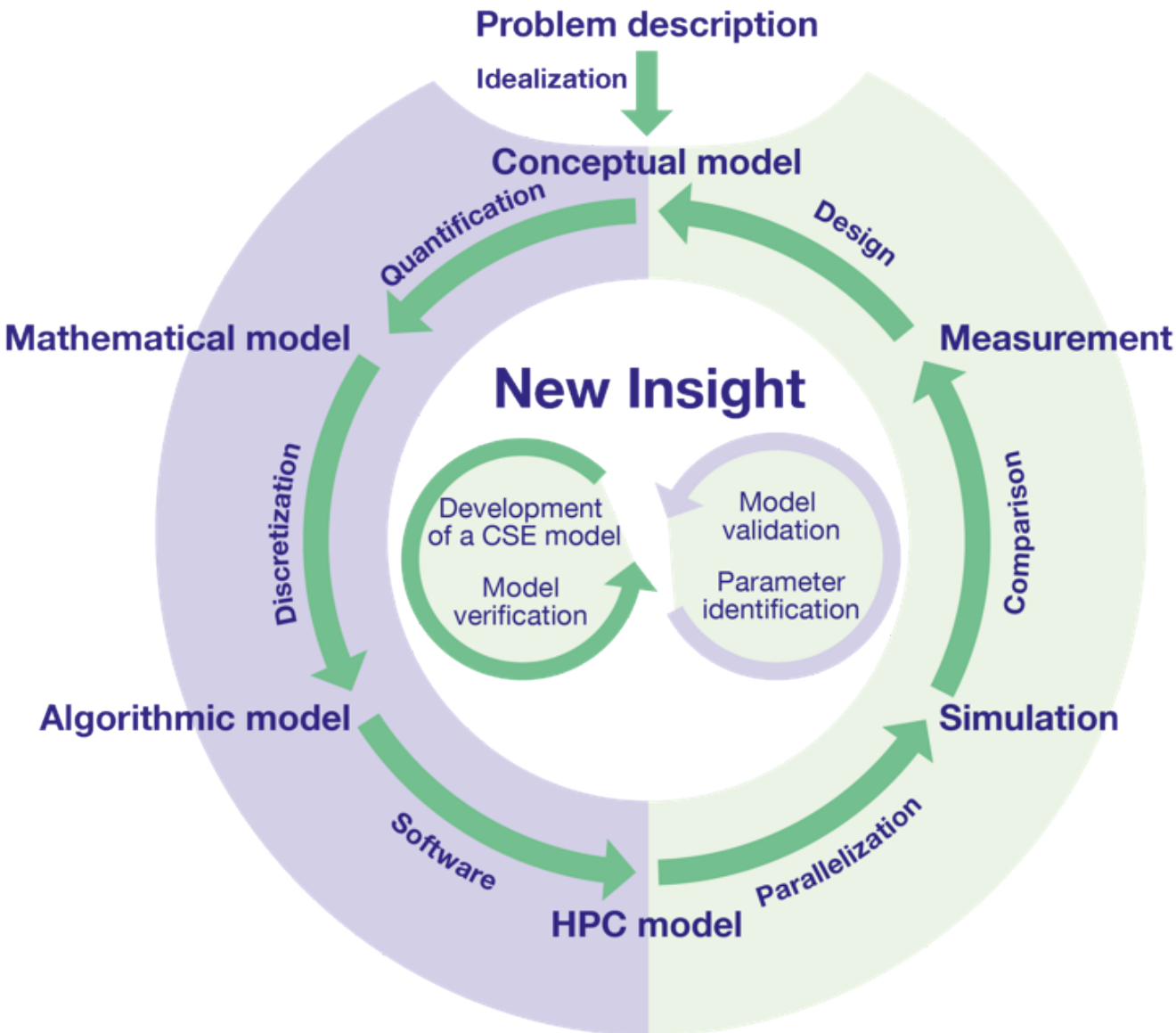
Note: Compiler technology

HPC "more than flops/second"

- Other architecture aspects: processor, communication, memory hierarchy, etc
- Overall computational environment: software, computer, grid, cloud

- *The relevant metrics for measuring speed is elapsed time from submission to completion of execution!*

The CSE Pipeline



The CSE Pipeline (cont)

1. Conceptual model (e.g., identification of physics...)
2. Mathematical model
3. Algorithmic model (often, numerical algorithm)
4. Computer code
5. Output, visualization
6. Validation, verification, feedback

Note: *3-6 are equally important for any HPC application!*

1 to 2

- Formulation of quantitative mathematical model (e.g., differential equation, integral equation, etc)
- Model derivation
  - Physically based modelling
  - Mathematical model reduction
  - Pre-determined model structure (e.g. neural nets)
- Analysis of models, existence, uniqueness, continuous dependence on data, consistency with respect to relevant properties (e.g. energy conservation)
- uncertainty quantification (UQ) — today often *in silico*
- *Matching model to computational resources*

The CSE Pipeline

1. Conceptual model (e.g., identification of physics...)
2. Mathematical model
3. Algorithmic model (often, numerical algorithm)
4. Computer code
5. Output, visualization
6. Validation, verification, feedback

2 to 3

- Formulation of a numerical algorithm that is appropriate for the mathematical model and the computational resources
- Derivation typically in two steps:
  - infinite to finite dimensional model (discretization)
  - algorithm for the finite dimensional model (linear system solver, Newton's method, multigrid etc)
- Build in adaptivity and error estimation
- Analysis of algorithm
  - Stability, accuracy, convergence etc
  - Consistent with special properties of the mathematical model
  - Computational complexity, fit to computer architecture

The CSE Pipeline

1. Conceptual model (e.g., identification of physics...)
2. Mathematical model
3. Algorithmic model (often, numerical algorithm)
4. Computer code
5. Output, visualization
6. Validation, verification, feedback

3 to 4

- Development of a computer code including libraries etc
- *Structure code and coding process for easy validation, debugging and collaborative work*
- Optimize message passing, threading and/or "help" the compiler to optimize cache handling and parallelization (e.g., automatic vectorization)
- Careful debugging of individual modules
- Reuse software, from BLAS and up

The CSE Pipeline

1. Conceptual model (e.g., identification of physics...)
2. Mathematical model
3. Algorithmic model (often, numerical algorithm)
4. Computer code
5. Output, visualization
6. Validation, verification, feedback

4 to 5

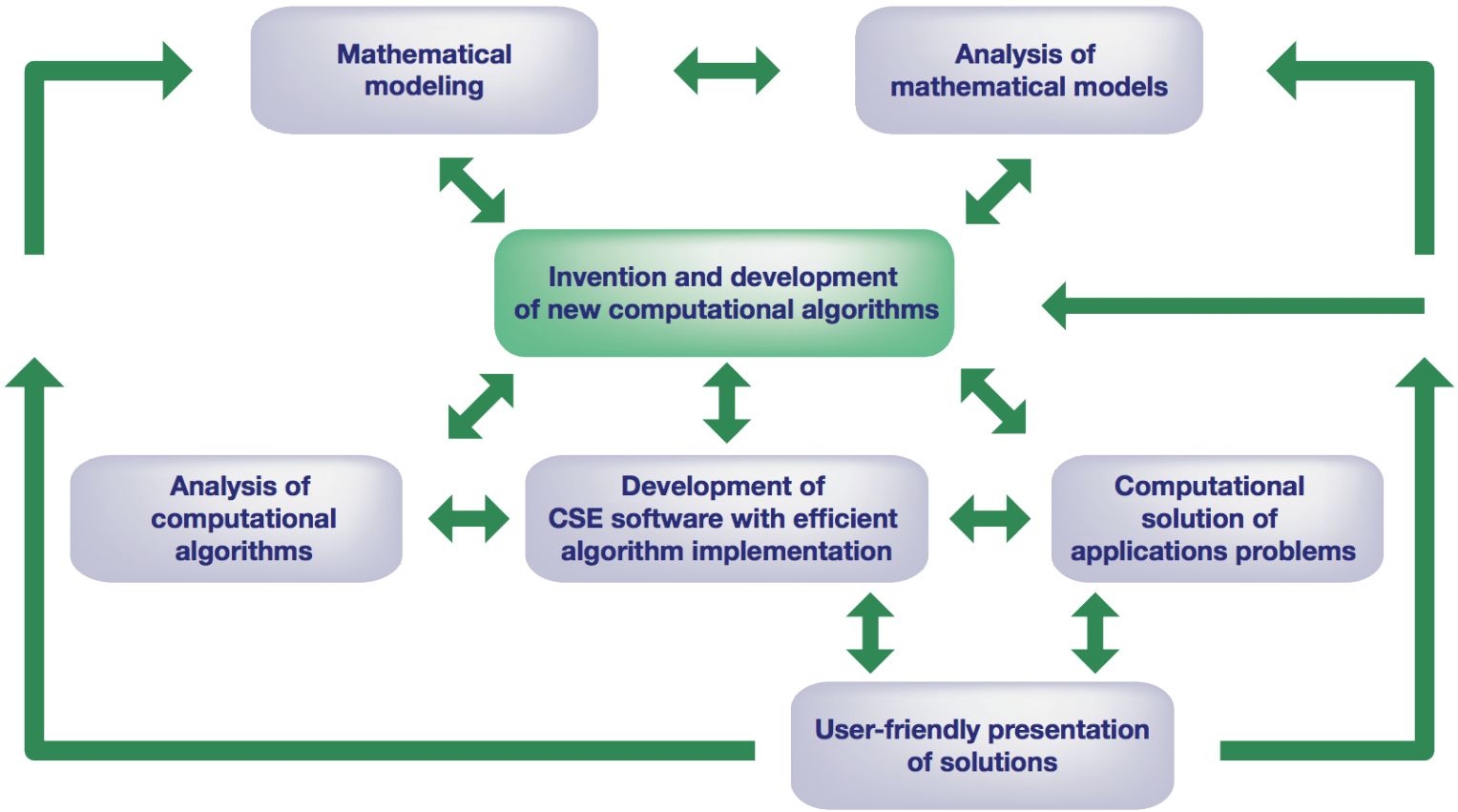
- Typically all done by the computer system
- Could include interactive steps of computational steering, collaborative work and interactive visualization
- Output could be input to other systems for further computation, e.g., optimization loop, model identification or control

- *Design output to support understanding of results and to aid in validation and debugging of the earlier steps*

The Complete Pipeline

- Verify that the code follows specifications
- Feedback to validate and optimize the computational pipeline. Check output with respect to
  - measured data
  - known model properties
  - results from known test cases and other codes
  - variation in parameters (e.g. mesh refinement)
- Find efficiency bottlenecks and try to eliminate them using all steps in the CSE pipeline

Interaction of CSE Components



General Remarks

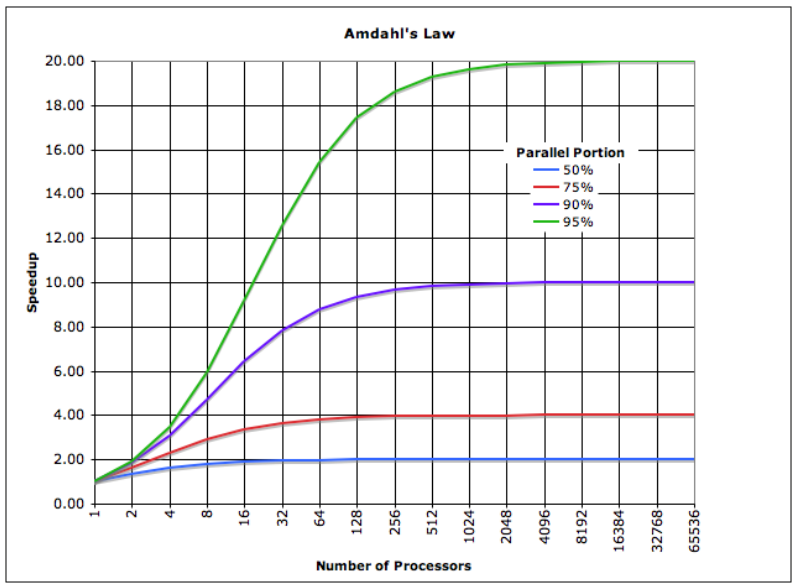
- The computational pipeline may be part of larger simulation, as i.e. the simulation step in an optimization
- Only part of pipeline may be relevant in a particular case as, i.e. in visualization of measured data
- Computations may be needed to define the mathematical model (*identification*)
- Strategy in development may vary
  - Will the code be used only once or thousands of times
  - Is the desired result goal oriented (i.e calculate drag of an airplane) or is the simulation for general discovery

General Remarks (cont)

- Two overarching goals: accuracy and computational efficiency
- Accuracy
  - Appropriate mathematical model
  - (Sufficiently) Accurate numerical algorithm (discretization, solver, truncation and round off errors)
  - Verification, validation, uncertainty quantification
- Computational efficiency – our focus here
  - Flops, load balance, communication, architecture dependencies

Time Sinks: Flops

- Algorithms with minimal number of flops (*often in conflict with algorithms that are easy to distribute*)
- Distribute flops to many processors
- Load balance for maximal use of processors: Amdahl's law. Speedup S_P on P processors is restricted by $S_P = P/(1+(P-1)f) < 1/f$ (f - sequential part)
- Data and operation flow (GPUs)



Time Sinks: Data Access

- Memory access time
  - Memory hierarchy, Cache strategy (depends on algorithm)
  - Pipelining of operations (prefetch, GPUs)
- Node to node communication
  - Use of parallelism in algorithms
  - Consider architecture of interconnect (i. e. multicore processors)
  - Consider both latency and bandwidth

Parallel Computing: The Beginning (1950)

Computational office at North American Aviation, Los Angeles

Communication: Increasing Importance

Annual improvement: Moore's law "predicts" 59% per annum. This is what we observe:

Time/flop
59%

	bandwidth	latency
network	26%	15%
DRAM	23%	5%

- Communication cost (time, energy) relative to cost for arithmetic growth for every new computer generation
- Exascale: paradigm shift time/flop << 1/bandwidth << latency
- Communication: L1 – L2 – DRAM – Network

Do Not Forget The Cost of Computing!

- In 2013, data centres in the US consumed 91 GWh electrical energy amounting to 2.4% of overall consumption
- Peak power consumption of US data centres 7 GW in 2007
- In comparison: Overall installed power capacity in Sweden 36 GW in 2013 (Germany: 153 GW)

Source: Center for Sustainable Systems, University of Michigan. 2017. "Green IT Factsheet." Pub. No. CSS09-07 and CIA World Factbook

The Situation in June 2018

Characteristics	R_{max} [TFlop/s]	Power [kW]	[GFlops/W]	Ext [MW]
Swedens fastest beskow (KTH)	1802	842	2	500
#1 Top500 Summit (US)	122300	8806	14	72
#2 Top500 Sunway Taihu (CN)	93014	15372	6	167
#1 Green500 Shoubu System B (JP)	858	47	18	56

Note. The column "Ext" is the virtual power consumption if a computer using the given technology would provide 1 EFlop/s.

Source: top500.org

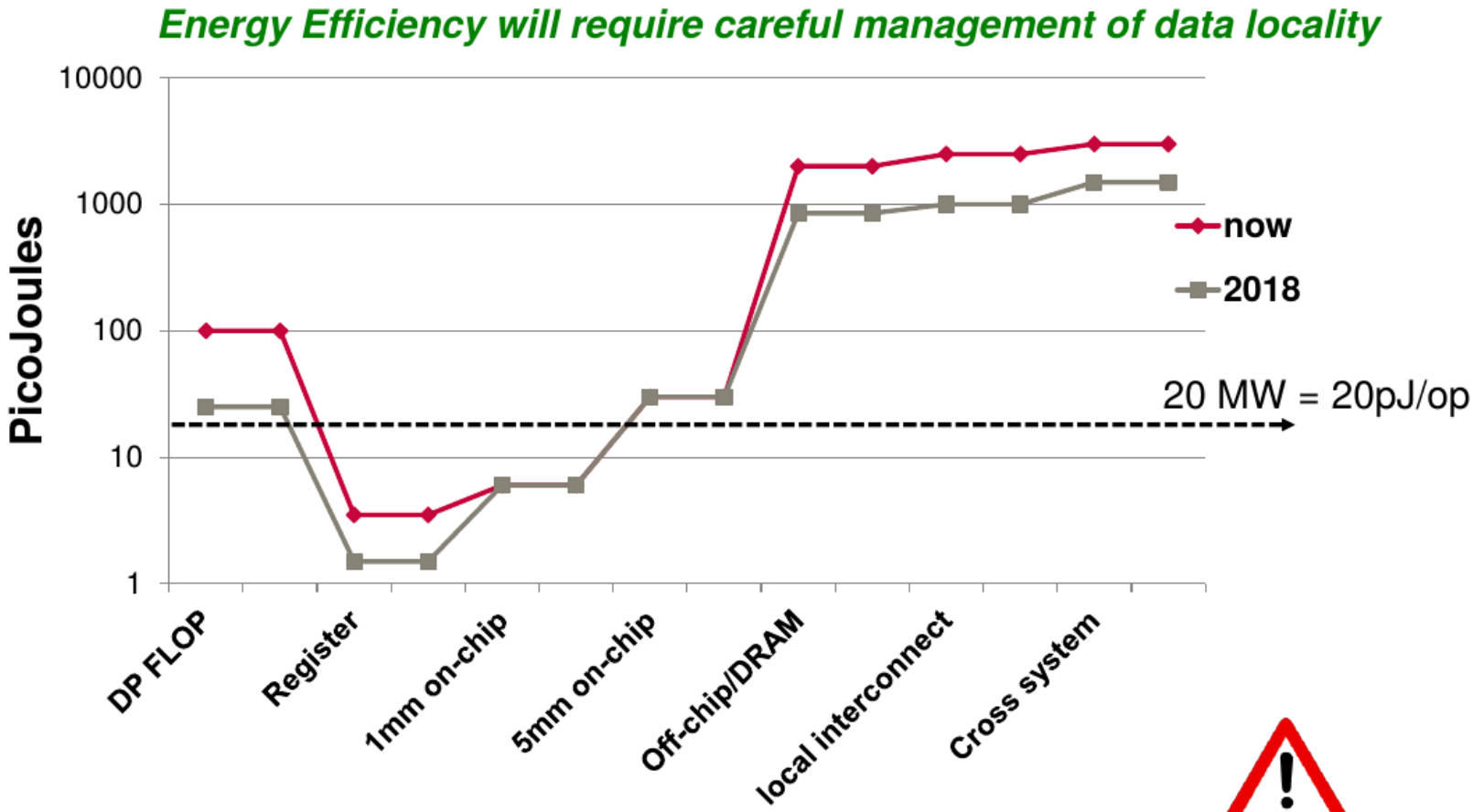
The Situation: A comparison

Average power consumption of a city with 100000 inhabitants (2013)

Country	Average power consumtion [MW]
US	920
Sweden	683
Germany	515
Japan	475
China	296
India	81

Source: Wikipedia, CIA World Factbook

Consequences: Communication and Exascale



Source : John Shalf/LBNL- SC'12

Mandatory to stay at socket/node level

Hardware Aspects for Efficient Algorithms and Software: Single Core

- Instruction pipelining
- Superscalar execution
- Vectorization
- Branches and branch prediction
- Cores may have their own cache memory
- Memory access delay
- Prefetching (uniform memory access)
- Multithreading (each core may need several threads to hide memory access latency)

Hardware Aspects: Node

- Several cores are on a chip
- Several cores may share cache memory
- Process affinity (context switching)
- Memory access bottlenecks may occur
- Memory pinning may be essential
- Nodes may be equipped with accelerators (e.g., graphic cards)
- Programming paradigm: suited for shared memory algorithms

Hardware Aspects: Cluster

- Thousands of nodes are connected by a fast network
- Different network topologies
- Often network has hierarchical structure itself
- High latencies require message aggregation
- Low bandwidth
- Pragramming paradigm: suited for distributed memory algorithms

Why Is Parallel Programming Interesting?

- A well behaved single processor algorithm may behave poorly on a parallel computer, and may need to be reformulated numerically
- There is no magic compiler that can turn a serial program into an efficient program all the time and on all machines
  - Performance programming involving low-level details: heavily application dependent
  - Irregularity in the computation and its data structures forces us to think even harder
  - Users don't start from scratch — they reuse old code. Poorly structured code, or code structured for older architectures can entail costly reprogramming

Simple Addition Example

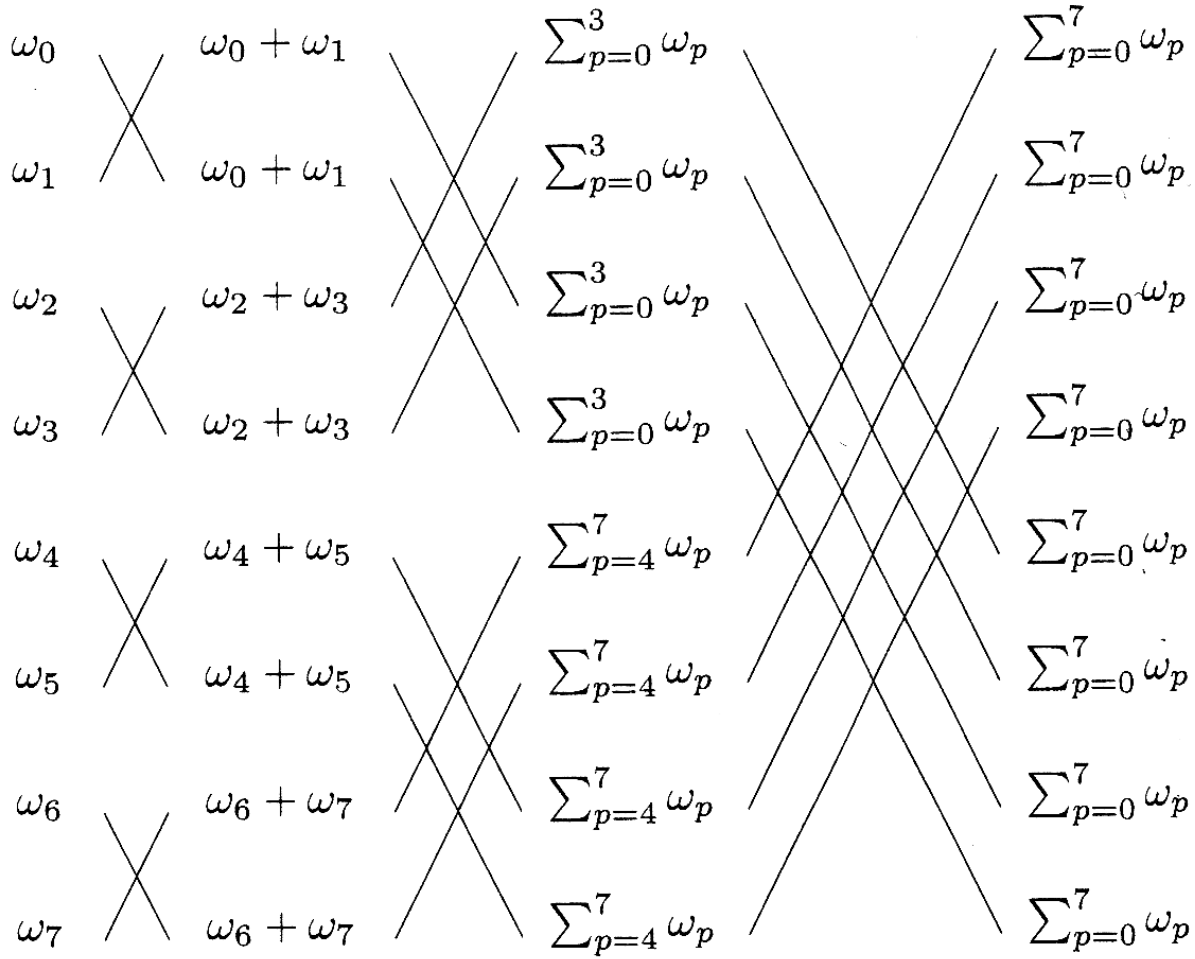
- A very large number N of values a_n shall be added on P processors:

$$S = \sum_{n=1}^{N} a_n = \sum_{p=0}^{P-1} \sum_{n \in I_p} a_n = \sum_{p=0}^{P-1} \omega_p$$

- Global summation: Example for $P = 8$

$$s = \underbrace{\underbrace{\underbrace{\omega_0 + \omega_1}_{} + \underbrace{\omega_2 + \omega_3}_{}}_{} + \underbrace{\underbrace{\omega_4 + \omega_5}_{} + \underbrace{\omega_6 + \omega_7}_{}}_{}}_{s}$$

Global Summation



Realization

- Assume that $P = 2^D$ is a power of 2. On process p, the program reads:

```
s = ω_p;
for d = 0:D-1
    q = bitflip(p,d);
    send(s,q);
    receive(h,q);
    s = s+h;
end
```

- The `bitflip` operation inverts bit number d in the binary representation of p.

Comments

- After execution of the program, *every* processor contains s
- Even if the number of flops is larger than theoretically necessary, the execution time compared to summation+broadcast is (much) shorter.

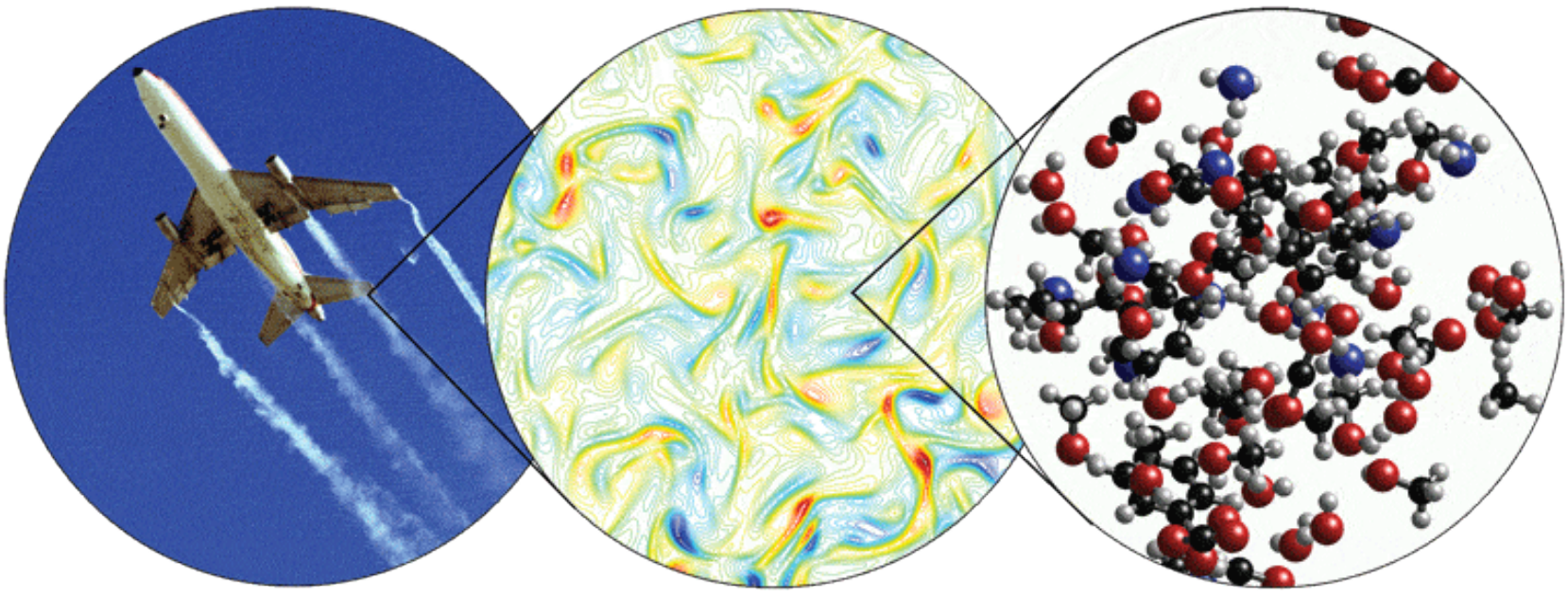
- Communication model: $t_{\text{comm}} = t_{\text{latency}} + wt_{\text{data}}$ (one message of length w)
- Execution time: $T_p = (2I_p - 1)t_a + \log P(t_{\text{latency}} + t_{\text{data}} + t_a)$
- Speedup

$$S_P \approx P \frac{1}{1 + \frac{P}{N} \log P \frac{t_{\text{latency}}}{t_a}}$$

- Good speedup only if $P \ll N$

Flops vs Accuracy

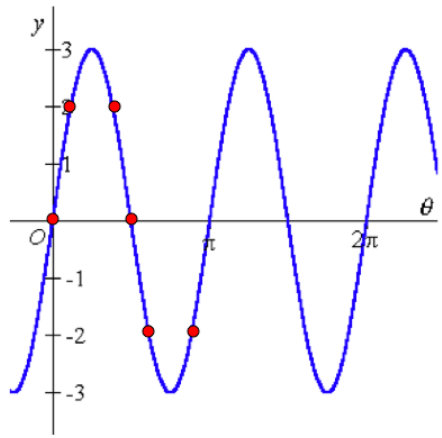
The main reasons for high computational cost (flops) in scientific computing are high dimensions and multi-scale phenomena. The smallest scales must be represented over the distance of the largest scales.



Flops vs Accuracy (cont)

Let the largest scale (or, wavelength of the lowest frequency) be 1 and the smallest scale be ε.Then:

- Let $N(\varepsilon, \delta)$ be the number of unknowns needed for a given accuracy in 1D:



  Typically, $N(\varepsilon, \delta) \sim C_\delta \varepsilon^{-1}$.

- If r denotes the computational cost per unknown (e.g. Gaussuian elimination gives $r = 3$, multigrid $r = 1$) and d the dimension of the problem, then

$$\#op = O(N(\varepsilon, \delta)^{dr}) = O(\varepsilon^{-dr})$$

- In the best case it holds $\#op = O(\varepsilon^{-d})$: atomistic simulation cannot be used directly for system scales.

Recommendations

- *The problem of large d and small ε must be handled already in the mathematical model.* Use effective or averaged equation whenever possible.
- For obtaining $r = 1$ (or close) use efficient methods as i.e. multigrid instead of Gaussian elimination. (Note that multigrid increases connectivity over simpler iteration algorithms and thus the communication cost in the simulation)
- An higher order numerical method (more accurate) requires lower N than a lower order in order to get the same accuracy in the result.

HPC Remarks

- For a given number of flops the overall computing time can be reduced by concurrent computing, load balancing, ordering and types of operations (* vs /), memory and communication strategies.
- Efficient distributed computation requires distributed algorithms *and sometimes even modified mathematical models*.
- The numerical algorithm also effects the possibility to write codes that optimally uses locality. *In many cases, more flops give less execution time!*
- All steps in scientific computing pipeline are coupled and must be handled as such!

Model and Algorithm: Effect on Parallelism

- Differential equations (local processes)
- Integral equations (global processes)
- Monte Carlo (direct simulation of stochastic processes)
- Optimization
- Sorting, searching, etc.
- Graph algorithms, etc.

You can never expect to construct an efficient software working against the physics of your problem!

Data Parallelism

- A large number of different data items are subjected to identical or similar processing all in parallel
- Example: rank sort

Data Partitioning

- Special type of data parallelism
- The data space is naturally partitioned into adjacent regions
- Each region is operated on in parallel by a different processor
- Examples: many numerical algorithms, image processing

Relaxed Algorithm

- Also known as *embarrassingly parallel*
- Each process computes in a self-sufficient manner with no synchronization or communication between processes
- Examples: rank sort, Monte Carlo algorithms, ray tracing

An Early Example of a Relaxed Algorithm

Veterans Bureau Calculations 1925(?)

Synchronous Iteration

- Each processor performs the same iterative computation on a different portion of data
- However, the processors must be synchronized at the end of each iteration
- Examples: many numerical algorithms

Replicated Workers

- A central pool of similar tasks is maintained
- A number of worker processes retrieve tasks from the pool
- The computations ends when the task pool is empty
- Examples: combinatorial problems, data base queries

Pipelined Computation

- The processes are arranged in a structure
- Each process performs a certain phase of the computation
- Example: microprocessors

Capturing the Physical Parallelism: Focus on Differential equations

- A large share of computations is devoted to solving differential equations.
- DEs have many applications in all fields of science and engineering.
- Properties:
  - Time: This is a sequential process (causality)
  - Space: concurrent processes (we can expect potential for parallelisation)
- Classification of algorithm according to different degrees of paralelism

ODEs: Initial Value Problems

- Typical applications: Molecular dynamics, electrical circuits, chemical reactions, astrophysics, rigid body dynamics
- Typical form

$$\frac{d}{dt}y = f(t, y), \quad y(t_0) = y^0$$

- *Causality is an obstacle to concurrent computing*

Difficulties in Parallelization

- Typical algorithm

$$y^n \approx y(t_n), \quad t_n = n\Delta t + t_0$$
$$y^{n+1} = F(y^n, y^{n-1}, \ldots, y^{n-r}, t_n), \quad n = r, r+1, \ldots$$

- Standard: sequential evaluation, no parallelism: y^n must be known before y^{n+1} can be computed.
- Options for parallelizations:
  - Parallel evaluation of F (efficient for large dimensions)
  - Special structure of F

Examples: Special Structure

$$\frac{d}{dt}y = f(t,y), \quad y(t_0) = y^0$$

(a) $f = \varepsilon g(y,t) + h(t)$

(b) $f = \varepsilon^{-1} g(y,t)$

(a) Very weak dependence on y (f mostly known)
(b) Very strong dependence on y (history not important - transients)

Special Structures (a)

- Rewrite the ODE as

$$y(t) = y_0 + \int_{t_0}^{t} h(\tau)d\tau + \varepsilon \int_{t_0}^{t} g(y(\tau), \tau)d\tau.$$

- Picard iteration will converge fast:

$$y^{(m+1)}(t) = y_0 + \int_{t_0}^{t} h(\tau)d\tau + \varepsilon \int_{t_0}^{t} g(y^{(m)}(\tau), \tau)d\tau$$

- Waveform relaxation: integrals can be evaluated in parallel, time interval can be split in subintervals. This leads to a pipelined parallelism.

Special Structures (b)

- Assume the ODE to be scalar, apply the implicit Euler method:

$$y^{n+1} = y^n + \Delta t \varepsilon^{-1} g(y^{n+1}, t_{n+1}),$$
$$y^{n+1} + \Delta t \varepsilon^{-1} g(y^{n+1}, t_{n+1}) = y^n$$

- The latter is a contraction for $\partial g(y,t)/\partial y < 0$ (the more "contracting" the smaller ε)
- Parareal algorithm: Multiple shooting with independent subintervals.
- This is a boundary value method applied to an initial value problem!

PDEs: Evolution Problems

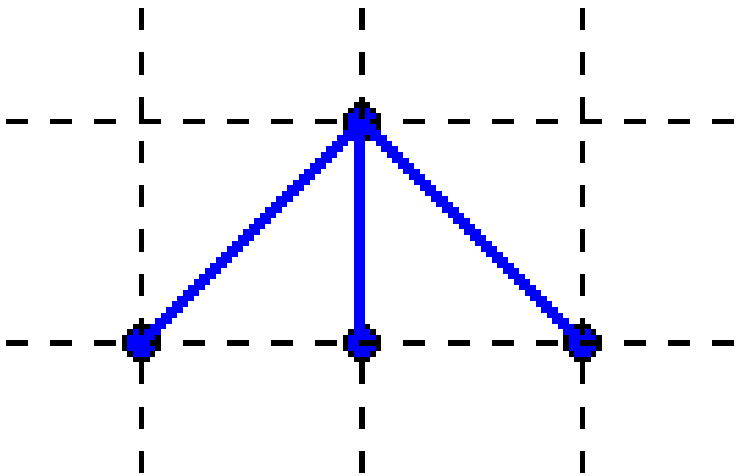
- Typical applications
  - All processes with local dependence
  - Examples: continuum and quantum mechanics, electromagnetics, meteorology, geophysics, financial models,..
- Typical form

$$\frac{\partial}{\partial t} u = f(\nabla_x, u, x, t) + IC + BC$$

- Natural concurrency in space: Domain Decomposition

First Generation Methods: Explicit in Time

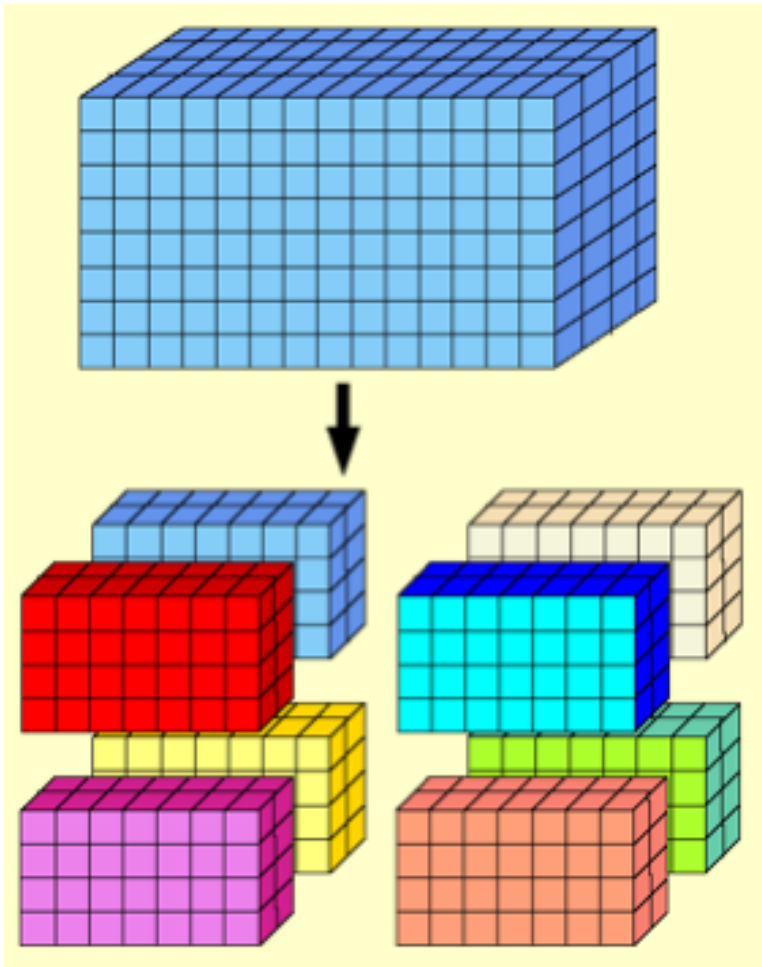
- A partial differential operator is local.
- Hence, apply a local discretization.
- Explicit: New grid values depend only on older neighbors:

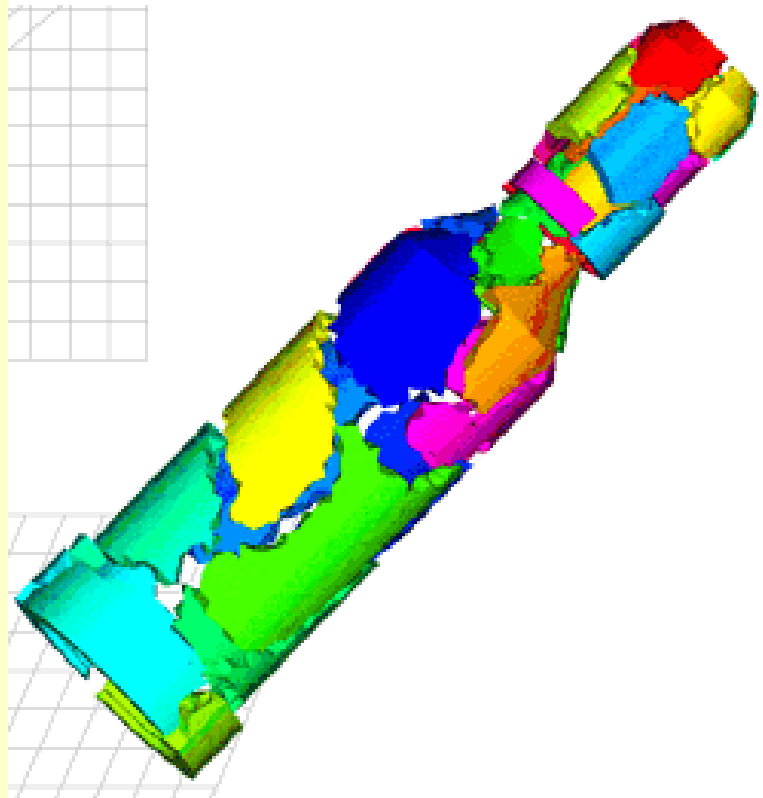


$$u_j^n \approx u(x_j, t_n)$$
$$u_j^{n+1} = F(u_{j+r}^n, \ldots, u_{j-r}^n, t_n)$$

Spatial Domain Decomposition

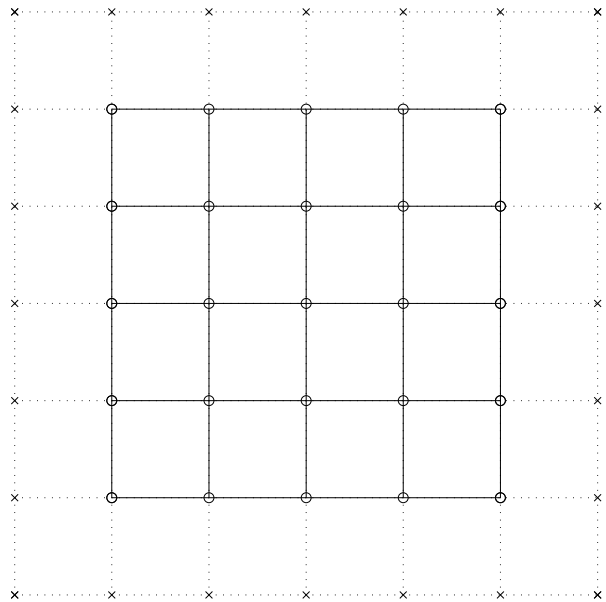
Distribute data (grid points, cells) to different processors





Domain Decomposition (cont)

- Each process needs values found on neighboring processes
- Use *ghost cells*,



  - Circles: local grid points
  - Crosses: ghost points

Domain Decomposition (cont)

- Scaling in 3D: number of interior points in block is $O(N^3)$
- number of block boundary points is $O(N^2)$
- $O(N^3)$ related to flops, $O(N^2)$ related to communication
- *High efficiency for large problem sizes*

- Overlapping DD for broader stencils and for multiple time steps between message passing (reduces latency effects)
- Further DD for reduced cache misses and for multicore
- For DD: connectivity in computational stencils is important not physical distance

Second Generation Methods: Implicit

- Explicit algorithm often have severe time step limitations due to stability requirements
- Implicit algorithms (a system of equations needs to be solved in each time step) typically have much less time step limitations
- Heat equation example:
  - explicit time step constraints $\Delta t \leq C \Delta x^2$
  - *implicit Crank-Nicolson has no constraints*

Implicit Algorithms

- The implicit step typically implies global coupling (all unknowns are coupled in each time step)
- *Efficient if the signal speed is high or infinite* (parabolic equations, hyperbolic multiscale equation, stiff problems)
- Similar solution strategy as in steady state problems (elliptic boundary value problems)
- Basic algorithmic component: fast parallel solver for systems of linear equations
  - Parallel Gaussian elimination (often in existing library)
  - As step in nonlinear iteration (Newton's method)
  - See also third generation iterative methods; multigrid, Krylov type methods

Stationary Problems, Elliptic Equations

- Stationary problems do not correspond to evolution processes (or evolution as time goes to infinity)
- Typical types of equations: elliptic equations, boundary integral equations, minimization problems
- Model problem: Laplace equation:

$$\nabla^2 u(x) = 0, \quad x \in \Omega$$
$$u(x) = g(x), \quad x \in \partial\Omega$$

Stationary Problems (cont)

- Discretization (FDM, FEM, quadrature,..) results in a linear or nonlinear system of equations
  - Differential equations: sparse systems
  - Integral equations: dense systems
- Existing software, for example ScaLAPACK (linear algebra software for distributed computing), requires special distribution of data, PBLAS, PETSc, etc.
- *Hot research topic*: rewrite linear algebra routines for reduced communication and new hardware architechtures

Gaussian Elimination

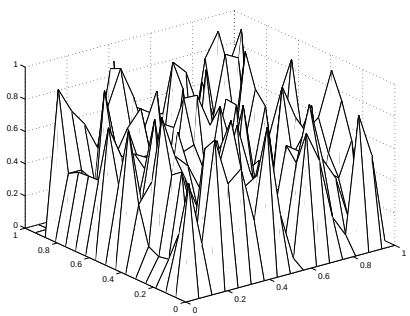
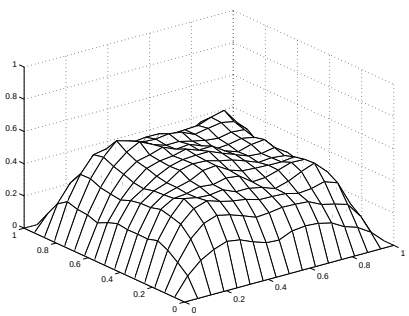
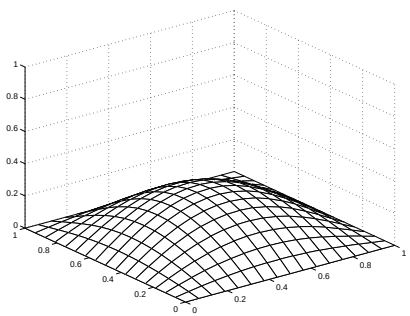
- Standard domain decomposition or block decomposition with regular Gaussian elimination leads to inefficient load balance.
- Choose a good pivoting strategy/data distribution (nontrivial!)
- *The time for backward subtitution may be longer than that of LU-decomposition!*

Third Generation Algorithms

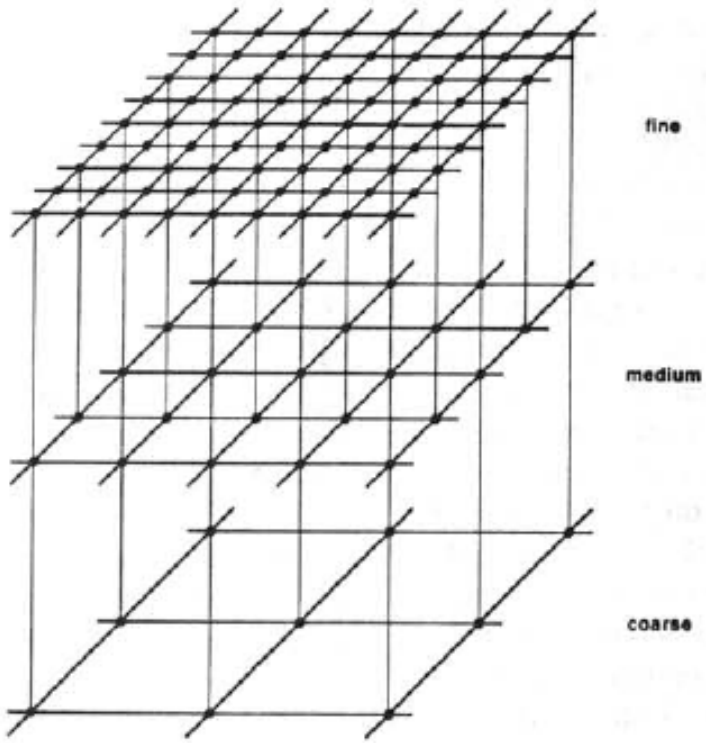
- First generation algorithms are easy to parallelize but may require many flops
- Second generation algorithms require coupling of all unknowns (solution of system of equations) at each time level
- Third generation algorithms introduces coupling in a more complex way (hierarchical methods)
  - They are constructed to provide an efficient global coupling
  - Examples: multigrid (MG), fast mulipole (FMM) and fast Fourier transform (FFT) methods

Towards Multigrid Methods

Application of Gauss-Seidel iteration to a discrete Poisson equation:
Error plot after 0, 3, 25 iteration

Multigrid Methods




```
procedure FMG
    u^{h_l} ← SOLVE (1, u^{h_l}, f^{h_l});
    for l ← 2 to L do
        begin
            v^{h_l} ← I_{l-1}^{l} u^{h_{l-1}};
            MG (l, v^{h_l}, f^{h_l})
        end;
```

```
procedure MG (l, u, g)
    if l = 1 then u ← SOLVE (1, u, g)
        else
            begin
                for i ← 1 to n_1 [while ... ] do u ← RELAX (l, u, g);
                v ← I_l^{l-1} u;
                d ← A^{h_{l-1}} v + I_l^{l-1} (g − A^{h_l} u);
                for i ← 1 to n_2 [while ... ] do MG (l − 1, v, d);
                u ← u + I_{l-1}^{l} (v − I_l^{l-1} u);
                for i ← 1 to n_3 do [while ... ] u ← RELAX (l, u, g)
            end;
```

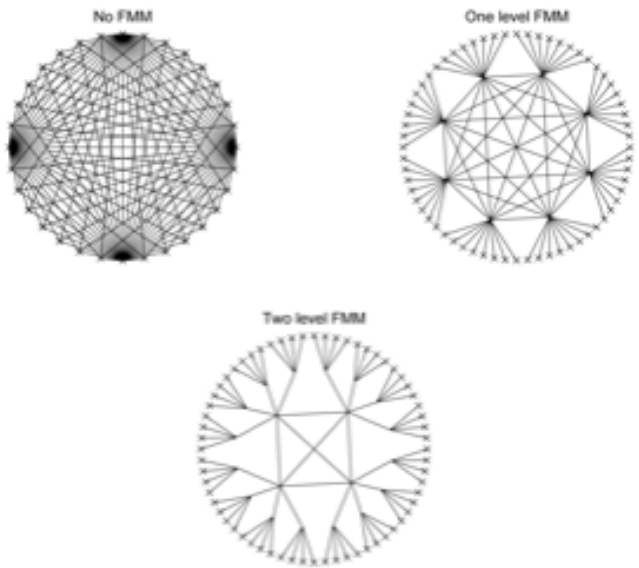
Multigrid Methods(cont)

- Efficient global coupling via interpolation to coarser grids
- Iteration with simple explicit local operator a few times on each grid level (compare explicit methods)
- Compute residual (error in equation) and use in correction at coarser grid level
- *From $O(N^3)$ (Gaussian elimination) to $O(N)$ computational complexity*

- Multigrid can also be used on unstructured grids and even on matrix problems without grids (algebraic multigrid)
- *Load balancing (Amdahl's law) and increased communication at coarse grid levels are difficulties in parallelization*

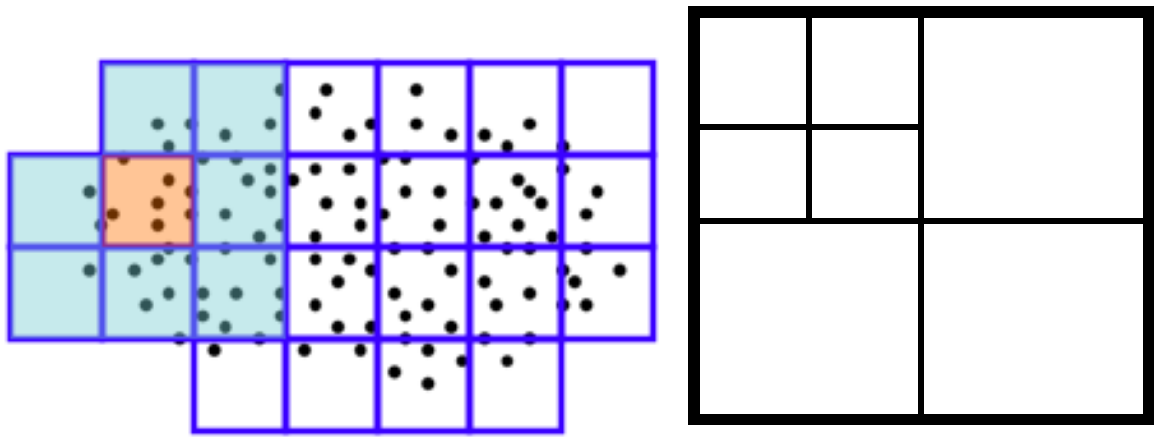
Fast Multipole Method (FMM)

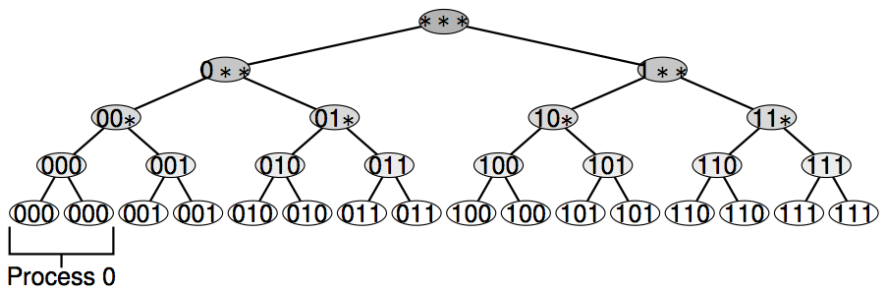
- Application area: Dense problems (boundary integral equations, molecular dynamics, similar)
- Point to point interaction requires $O(N^2)$ operations. FMM reduces the computational complexity to $O(N)$
- *Can be seen as fast matrix-vector multiply* (low-rank approximations: $A = BCD$, $\dim C \ll \dim A$)
- Simplified far field interaction: Coulomb forces



FMM (cont)

- Simplified far field interaction – compression
- Near field: explicit algorithm





Fast Fourier Transform (FFT)

$$c_k = \frac{1}{N} \sum_{j=0}^{N-1} \omega^{jk} f_j, \quad \omega = e^{2\pi i/N}$$

- FFT (Cooley and Tukey) is a divide-and-conquer algorithm: A DFT of dimension N corresponds to a combination of two DFTs of dimansion $N/2$.
- *FFT reduces the* $O(N^2)$ *computation to* $O(N \log N)$
- Multidimensional FFT quite difficult to parallelize
- Typically, there exists efficient software (FFTW)
- Dense matrix multiply can be seen as a product of sparse matrix multiplies

$$c = Wf = W_1 W_2 \cdots W_J f$$

Summary

- Consider all steps in the CSE pipeline for validation and computational efficiency
- Consider potential concurrency and locality in the physical and mathematical models when designing the parallel computational algorithm
- Balance the potential for efficient parallel implementation of simple numerical methods versus the reduced number of flops of more complex numerical methods.
- In algorithm design use simple model for latency-bandwidth-flop ratios, load balancing, memory access cost, etc.