Algorithms and Data Structures for Scalable Concurrent Programs

Björn Engquist

- Scientific computing: overview and HPC aspects of scientific computing
- Scalable concurrent programs: relation between the original physical problem and distributed algorithms

High Performance Computing, PDC Summer School, KTH, 2010

Scientific Computing

- Computing for science and engineering
- Based on applied fields of science, mathematics, computer science
- Typically requires HPC – infinite dimensional problems approximated by finite dimensional models
- Other terminologies for "scientific computing": "numerical analysis", "virtual prototyping", "computational science and engineering"

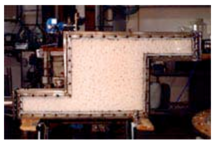
Numerical analysis

- Original term used since the early 1950s
- Today it denotes
  - Development of numerical algorithms from given mathematical models
  - Analysis of these algorithms: consistency, stability, convergence, order of accuracy, computational complexity

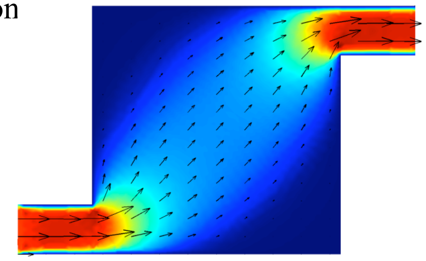
$\left(Example, error\ estimate: \left|u(x_j) - u_j\right| \le Ch^p \quad j = 1,2,..J\right)$

Virtual prototyping

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



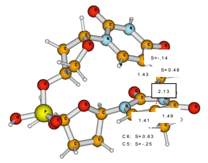
- Understanding
- Verification
- Planning, optimization and control



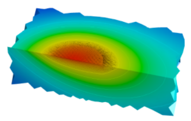
Example from paper industry

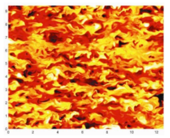
Computational science

The expression computational science is broader and includes simulations for scientific discovery.



Examples from life sciences (molecular biology), Material science (phase transition in welding), fluid dynamics (turbulence)

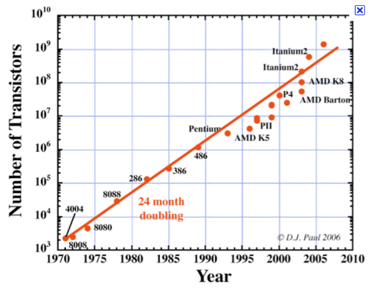


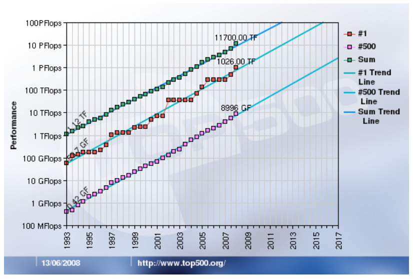


Level of peta-scale computing: a break through for computational science

Evolution of scientific computing

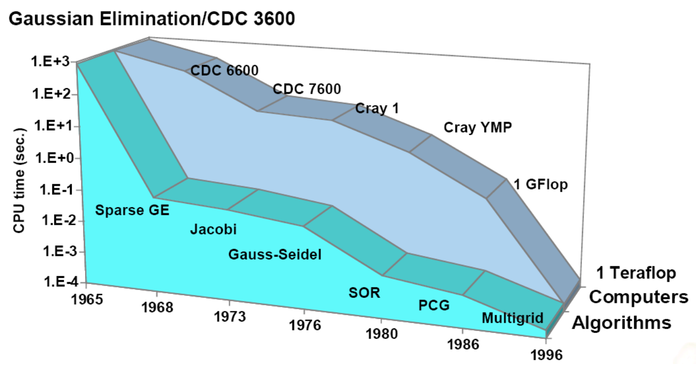
The development of scientific computing is based on progress in computer technology (Moor's law)





Evolution of scientific computing

The development of scientific computing is based on progress in computer technology (Moor’s law), but also on algorithms and software.



HPC “more than flops/second”

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

“***Slow Moving Clouds Fast Enough for HPC***
Ian Foster penned an interesting blog comparing the utility of a supercomputer to that of public cloud for HPC applications. Foster pointed out that while the typical supercomputer might be much faster than a generic cloud environment, the turnaround time might actually be much better for the cloud. He argues that "the relevant metric is not execution time but elapsed time from submission to the completion of execution."

The Scientific Computing Pipeline

$$\begin{tikzcd}
\text{1. scientific process} \arrow[d] & \\
\text{2. mathematical model} \arrow[d] & \\
\text{3. numerical algorithm} \arrow[dd] & \text{validation} \\
 & \text{verification} \\
\text{4. computer code} \arrow[d] & \text{feedback} \\
\text{5. output, visualization} \arrow[ru] &
\end{tikzcd}$$

$1 \rightarrow 2$

- Formulation of quantitative mathematical model (i.e. differential equation, integral equations, etc.)
- Model derivation
  - Physically based modelling
  - Mathematical model reduction
  - Pre-determined model structure (i.e. neural nets)
- Analysis of models, existence, uniqueness, continuous dependence on data, consistency with respect to relevant properties (i.e energy conservation)
- Matching model to computational resources

$2 \rightarrow 3$

- Formulation of numerical algorithm that is appropriate for the mathematical model and the computational recourses
- Derivation typically in two steps:
  - infinite to finite dimensional model (FDM, FEM,..)
  - algorithm for the finite dimensional model (Gaussian elimination, Newton's method, multigrid etc.)
- Build in adaptively and error estimation
- Analysis of algorithm
  - Stability, accuracy, convergence, etc.
  - Consistent with special properties in mathematical model
  - Fit to computer architecture

$3 \rightarrow 4$

- 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" compiler to optimize cache handling and parallelization
- Careful debugging of individual modules
- Reuse software, from BLAS and up
- Consider grid aspects

$4 \rightarrow 5$

- 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, i.e. optimisation loop, model identification, or control
- Design output to support understanding of results and to aid in validation and debugging of the earlier steps 1 to 4

$1 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 4 \leftrightarrow 5$

- 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 - i.e. mesh refinement
- Find efficiency bottlenecks and try to eliminate them using all steps in the scientific computing pipeline

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

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
- Memory access time

  - Cache strategy (depends on algorithm)
  - Memory hierarchy
  - 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

Simple addition example

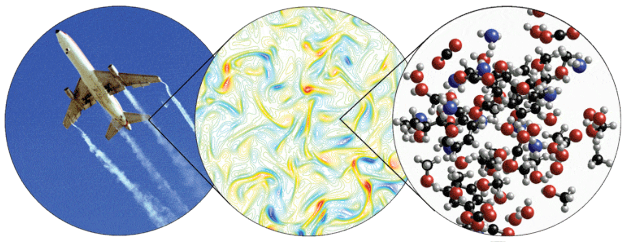
- Very large sum (N numbers) with access to very large number of processors $S = \sum_{n=1}^{N} a_n$
- Flop time δ_1 , communication time (one number) δ_2
- Distribute m numbers to each processor
- Cost of summing including communication (one step)

$$time = (m + (N/m))\delta_1 + (N/m)\delta_2$$

- Discuss following steps and optimal choices of m-values

Computational complexity

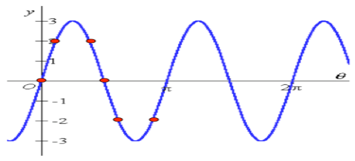
The main reasons for high computational cost (flops) in scientific computing are high dimensions and multi-scale phenomena. The smallest scales must be represented over the distance of the largest scales



If 1 is the largest scale (or wave length of the lowest frequency) in each dimension and ε is the smallest scale then,

$$\text{Flops} = O(N(\varepsilon, \delta)\ \varepsilon^{-1})^{dr})$$

- $N(\varepsilon, \delta)$ is the number of unknowns needed for a given accuracy δ. Typically $2 < N < C_\delta \varepsilon^{-1}$. Methods with higher order accuracy gives lower N.



- d is the number of dimensions
- r measures the computational cost per unknowns. Explicit methods: r=1, Gaussian elimination of a dense matrix: r=3.

- Best possible case: flops=$O(\varepsilon^{-d})$ $\rightarrow$ "atomistic simulation can not be used directly for system scales"
- Compare Shannon sampling theorem: "a signal requires at least two points /wavelength to be exactly represented"

Reduction of flops

- The problem of large d and small ε must be handled already in the mathematical model. Use effective or averaged equation whenever possible.
- For $r \rightarrow 1$ 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, 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 cache (localization).
- All steps in scientific computing pipeline coupled

Model and Algorithm: effect on parallelism and localization

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

Capturing the physical parallelism in modern computer architectures

- Discretization of differential equations
- Time: sequential processes (causality)
- Space: concurrent processes
- Classification of algorithms – different degrees of concurrency

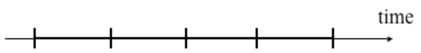
ODEs: initial value problems

- Typical applications
  - Molecular dynamics
  - Chemical reactions
  - Astrophysics
  - Rigid body dynamics
  - Electrical circuits
- Typical form $\frac{dy}{dt} = f(y,t), \quad y(t_0) = y_0$
- Causality: obstacle to concurrent computing

Difficulties in parallelization

- Typical algorithm

$$y^n \approx y(t_n), \quad t_n = n\Delta t$$



$$y^{n+1} = F(y^n,..,y^{n-r},t_n), \quad n = r, r+1, r+2,..$$

- Standard: sequential evaluation - no parallelism: y^n must be known before y^{n+1} is calculated
- Options for parallelization
  - Parallel evaluation of F - efficient for very large u and F dimensions (example, molecular dynamics)
  - Special structure of F

Special structure of *f* and *F*

$$\frac{dy}{dt} = f(y,t), \quad y(t_0) = y_0, \quad 0 < \varepsilon << 1$$
$$(a) \quad f = \varepsilon g(y,t) + h(t)$$
$$(b) \quad f = \varepsilon^{-1} g(y,t)$$

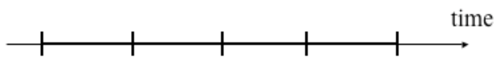
(a) "Very weak dependence on *y* (history mostly known)"

(b) "Very strong dependence on *y* (history not so important)"

Special structure, cont.

$$(a) \quad y^{(m+1)}(t) = y_0 + \int_0^t h(\tau)d\tau + \varepsilon \int_0^t g(y^{(m)}(\tau),\tau)d\tau$$

- Iterate over m - compare Picard iteration (small ε means fast convergence
- The integrals can easily be evaluated in parallel, time segment decomposition, waveform relaxation



Special structure, cont.

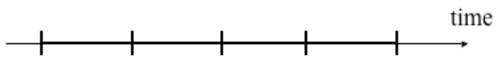
- Type (b) with the implicit Euler method (scalar)

$$\frac{dy}{dt} = f(y,t), \quad y(t_0) = y_0, \quad (b) \quad f = \varepsilon^{-1} g(y,t)$$

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

$$(y^{n+1} - \Delta t \varepsilon^{-1} g(y^{n+1}, t_{n+1})) = y^n, \quad \frac{\partial g(y,t)}{\partial y} < 0 \quad \Rightarrow \quad \textit{contraction}$$

- Also interval decomposition. Initial interval points unknowns in outer iteration.



PDEs: initial and initial/boundary value problems (evolution)

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

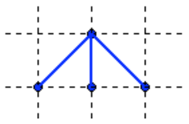
$$\frac{\partial u}{\partial t} = f(\nabla_x, u, x, t), \quad u(x, t_0) = u_0(x) \textit{ and BCs}$$

- Natural concurrency in space → domain decomposition (sequential in time)

Explicit methods (first generation algorithms)

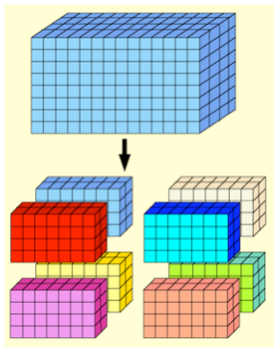
- A partial differential operator is local → natural with local discretization. New grid value depends on older neighbors.

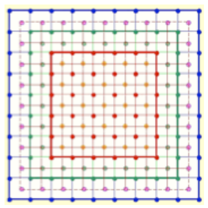
$$u_j^n \approx u(x_j, t_n)$$
$$u_j^{n+1} = F(u_{j+r}^n, .., u_{j-r}^n, x_j, t_n)$$

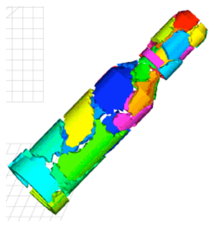


Spatial domain decomposition (DD)

- Distribute data (grid points, cells) to different processors

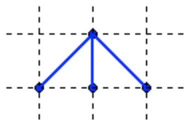






Remarks: DD

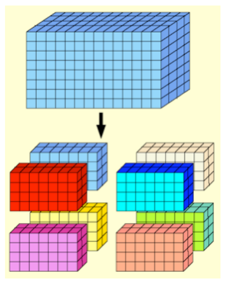
- Further DD for reduced cache misses and for multicore.
- For DD: connectivity in computational stencils is important not physical distance
- Overlapping DD for broader stencils and for multiple time steps between message passing (reduces latency effects)



Explicit computational difference stencil

Remarks: DD

- Scaling: number of interior points in block ≈ O(N^3)
  number of block boundary points ≈ O(N^2)
- O(N^3) related to flops, O(N^2) related to communication
- High efficiency for large problem sizes
- Consider band width versus latency in decomposition strategy

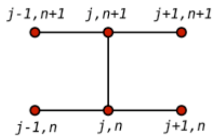


Implicit algorithms (second generation algorithms)

- 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: no constraints

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



Implicit algorithms, cont.

- 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, boundary value problems, elliptic equations

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

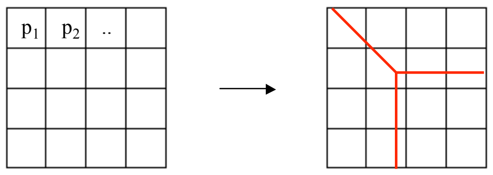
$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0, \quad (x,y) \in \Omega \; (domain)$$
$$u(x,y) = f(x,y), \quad (x,y) \in \partial\Omega \; (boundary)$$

Stationary problems, continued

- 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, requires special distribution of data (ScaLAPACK: linear algebra software for distributed computing).
- Natural domain distribution of data may not be optimal for parallel computing. Example Gaussian elimination

Gaussian elimination

- Standard domain decomposition or block decomposition with regular Gaussian elimination leads to inefficient load balance
- → redistribute, for example, the rows (classical example of parallel algorithm)

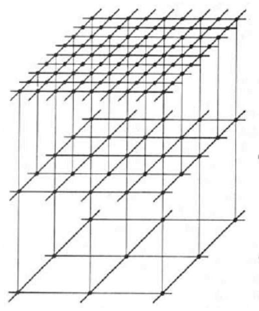


Third generation algorithms

- 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
  - Examples, multigrid and Krylov subspace methods for solution of systems of equations
  - Fast mulipole and fast Fourier transform methods

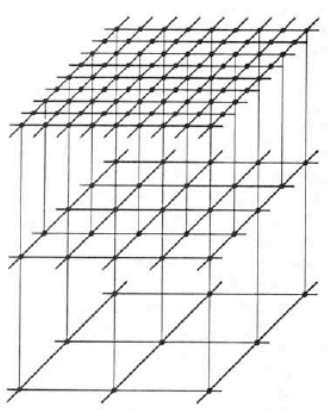
Multigrid

- Efficient global coupling via interpolation to coarser grids
- From O(N^3) (Gaussian elimination) to O(N) computational complexity



```
procedure FMG
    $u^{h_1} \leftarrow$ SOLVE (1, $u^{h_1}$, $f^{h_1}$);
    for $l \leftarrow 2$ to $L$ do
        begin
            $v^{h_l} \leftarrow I_{l-1 \to l} u^{h_{l-1}}$;
            MG ($l$, $v^{h_l}$, $f^{h_l}$)
        end;

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

Multigrid



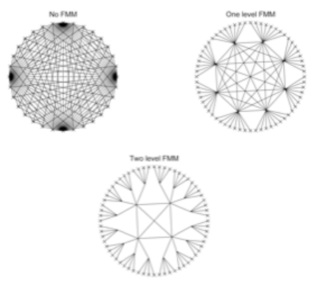
```
procedure FMG
    $u^{h_1} \leftarrow$ SOLVE (1, $u^{h_1}$, $f^{h_1}$);
    for $l \leftarrow 2$ to $L$ do
        begin
            $v^{h_l} \leftarrow I_{l-1 \to l} u^{h_{l-1}}$;
            MG ($l$, $v^{h_l}$, $f^{h_l}$)
        end;

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

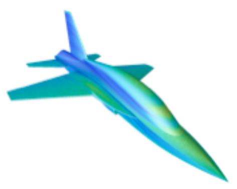
Multigrid, cont.

- Iteration with simple explicit local operator a few times on each grid level
- Compute residual (error in equation) and use in correction at coarser grid level
- 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 grids are difficulties in parallelization

Fast multipole method (FMM)

- Point to point interaction requires O(N^2) operations. FMM reduces the computational complexity to O(*NlogN*)
- Can be seen as fast matrix-vector multiply
- Simplified far field interaction - compare gravity



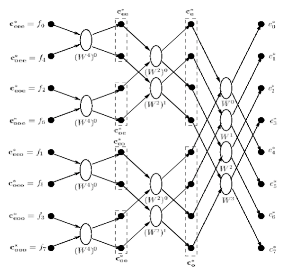


Electromagnetic example

Fast Fourier transform (FFT)

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

- FFT also reduces an O(N^2) computation to O($NlogN$)
- Multidimensional FFT quite difficult to parallelize
- Typically, there exists efficient software



- Dense matrix multiply → product of sparse matrix multiplies

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

Summary

- Consider all steps in the scientific computing 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.