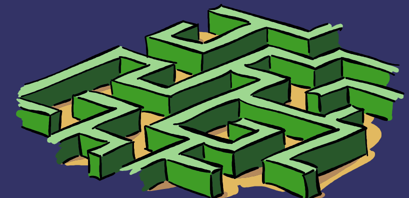
Scientific computing applied to linear algebra

Philippe Caussignac

Jaroslaw Glowacki

Reliability, precision

High Performance Computing



Reliability, precision

- Programming language with types, strict syntax , compiled program, debugger
- Definition of data structures, modular programming
- Errors in algorithms programming
- Roundoff errors
- Compiler bug, machine bug
- Wrong model

High performance computing

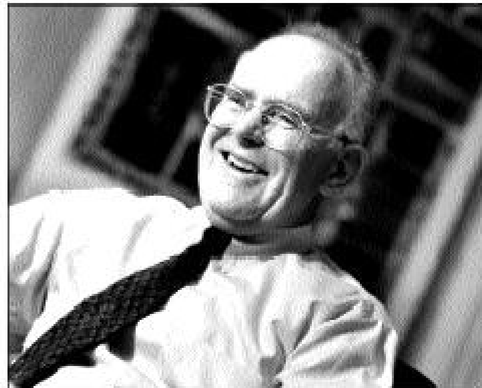
- Why?

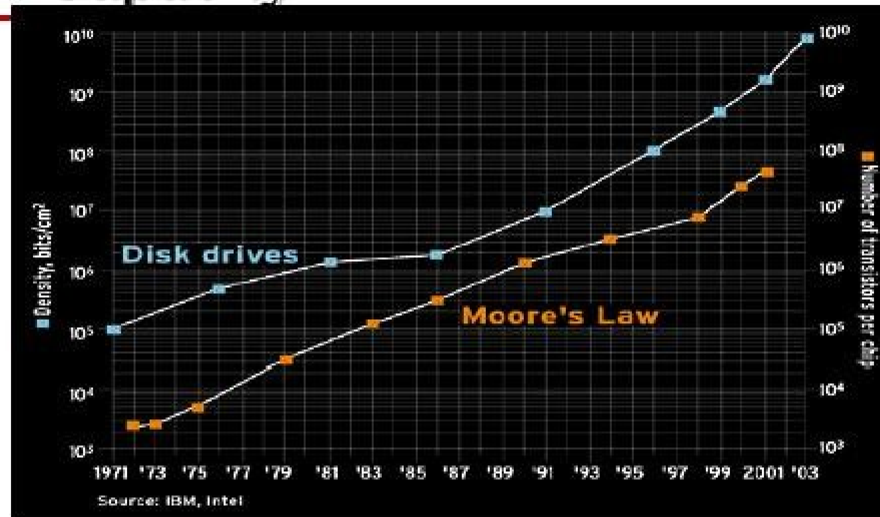
- Big amount of data (example: database)

- Big amount of operations (example: Monte-Carlo methods, molecular dynamics)

- Big amount of operations together with a big amount of data (example: solution of PDE's)

Technology Trends: Microprocessor Capacity





Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

2X transistors/Chip Every 1.5 years
Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful. Not just processors, bandwidth, storage, etc.
2X memory and processor speed and ½ size, cost, & power every 18 months.

Moore's Law

Super Scalar/Vector/Parallel

1 PFlop/s
1 TFlop/s
1 GFlop/s
1 MFlop/s
1 KFlop/s

Parallel
Vector
Super Scalar
Scalar

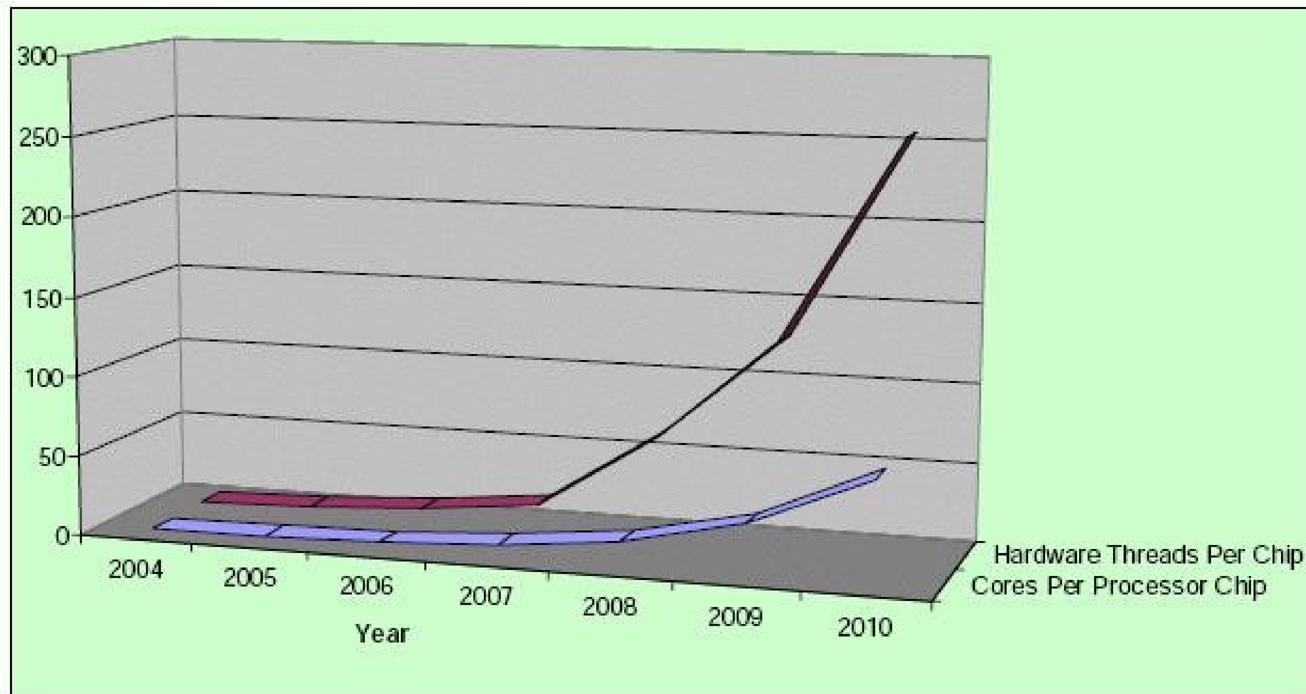
Earth Simulator
ASCI White Pacific
ASCI Red
TMC CM-5
Cray T3D
TMC CM-2
Cray 2
Cray X-MP
Cray 1
CDC 7600
IBM 360/195
CDC 6600
IBM 7090
UNIVAC 1
EDSAC 1

1950 1960 1970 1980 1990 2000 2010

1941 1 (Floating Point operations / second, Flop/s)
1945 100
1949 1,000 (1 KiloFlop/s, KFlop/s)
1951 10,000
1961 100,000
1964 1,000,000 (1 MegaFlop/s, MFlop/s)
1968 10,000,000
1975 100,000,000
1987 1,000,000,000 (1 GigaFlop/s, GFlop/s)
1992 10,000,000,000
1993 100,000,000,000
1997 1,000,000,000,000 (1 TeraFlop/s, TFlop/s)
2000 10,000,000,000,000
2003 35,000,000,000,000 (35 TFlop/s)

CPU Desktop Trends 2004-2010

- Relative processing power will continue to double every 18 months
- 256 logical processors per chip in late 2010



Content

1. Numerical simulation: coding, languages, sources of errors
2. Principles of software validation
3. Stability, computability, reliability
4. Data perturbations: theory and practice
5. Computer types, performance
6. Basic operations in linear algebra
7. Linear systems with dense matrix, LAPACK
8. Linear systems with sparse matrix
9. Message passing, MPI
10. Linear systems on DMM, Scalapack
11. Iterative methods

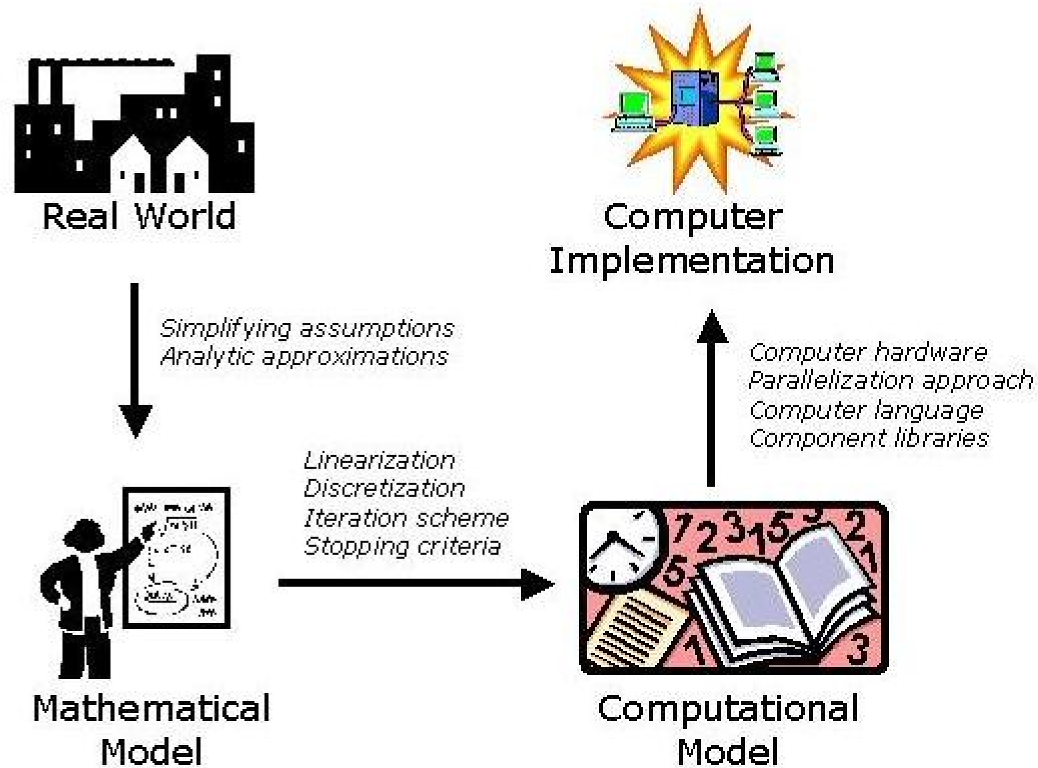
Goals

- Awareness of reliability and stability problems
- Identify the results of various error types
- Learn good principles of software design in scientific computing

- Awareness of the important problems in high performance computing for scientific applications
- Basic libraries (Blas, Lapack,...)

- Parallel algorithms
- Tools for parallel programming (OpenMP, MPI)
- Libraries for parallelizing (Scalapack,...)

Numerical simulation



Coding principles

Operators and data structures close to the mathematical model:

```
A(n,n), x(n), b(n), x=solve(n,A,b)
```

Top-down and bottom-up programming:

```
A=LU,  Ly=b,  Ux=y
 s=(x,y),  x=a*x+y
```

Use of standard libraries, principally at the lowest level

Commented software, readable by someone else

Programming language for scientific computing (According to PhC)

Candidates : Fortran (f77,f90), C, C++, Java

(No more Pascal compiler)

Absolutely required

Fast arithmetic operations (excludes Java?)

Types (structures), dynamic memory (excludes f77)

Portability, free compiler (excludes f90, gfortran?)

Wished: objects

Templates, encapsulation, overloading

Too many problems with C++ to force some default behaviours of the compiler

Conclusion: C is the good choice

Compilation, linking

Compilation: translation of language instructions to machine language (objects)

`gcc -c -O3 main.c func.c`

`creates` the objects `main.o func.o`

Linking: adds the external references to the objects (I/O, math. functions, specific libraries)

the program (loader) is named ld; run it through the compiler:

`gcc -o prog main.o func.o mylib.a -lblas`

`creates` the executable `prog`

`mylib.a objects` library in the local directory

`-lblas libblas.a in the default directories (/usr/lib /usr/local/lib ...)`

Useful options

`-g` for debugging

Then run **gdb** or **ddd**

```
-L library_path
-I  include_file_path
```

Other compiler: Intel

```
/opt/intel/cc/9.1.045/bin/icc
```

Makefile (cf. Exercises)

```
make [-f makefile_name]
```

Sources of errors

Wrong mathematical model
Inefficient or unstable algorithm

Syntax error
Overflow, memory overflow, ...

Error in algorithm coding

Error due to parallelization

Error of the floating point arithmetics

Computer Arithmetics

Real numbers approximated by floating point numbers

$$f = \pm m \cdot b^{\pm e}$$

Addition: normalization with b=10, 7 digits mantissa

$$r_1 = 1,\ \ r_2 = 10^{-7}\ \ \Rightarrow$$

$$f_1 = 0.100000 \cdot 10^{+01,}\ \ f_2 = 0.1000000 \cdot 10^{-06}$$

$$g_1 = 0.1000000 \cdot 10^{+01,}\ \ g_2 = 0.0000000 \cdot 10^{+01}$$

Roundoff error:

$$x \in \mathbb{R} \simeq x_\epsilon \in IF \Rightarrow (1.0)_\epsilon + \epsilon/2 = (1.0)_\epsilon,\ \ \epsilon \simeq 10^{-16}$$

Other errors

Cancellation

$$C\ float:$$
$$x^2 - bx + c = 0, b = 742.0, c = 2.0$$
$$\Delta = \sqrt{(b^2 - 4c)} = 741.9946, x_1 = (b + \Delta)/2 = 741.9973, x_2 = (b - \Delta)/2 = 0.002686$$
$$x_1 + x_2 = 742.0, x_1 * x_2 = 1.9927$$
$$x_1 = (b + \Delta)/2 = 741.9973, x_2 = c/x_1 = 0.002695$$
$$x_1 + x_2 = 742.00001, x_1 * x_2 = 2.0$$

Accumulation (recursion)

$$n = 10000;\ x = 0.0;\ h = 1.0/n;$$
$$while\ (x \leq 1)\ x = x + h;$$
$$Result: x = 1.000054$$

Integer overflow: no error message

$$\text{int}\ i, n = 20, nfact = 1;$$
$$for\ (i = 1; i \leq n; i{+}{+})\ nfact *= i;$$
$$Result: n! = -2102132736 \quad (n \leq 12 \quad OK)$$

http://www5.in.tum.de/~huckle/bugse.html

Main coding pitfalls

Constants with infinite number of digits: $\pi = 3.14,...$

Test on equality between real numbers

Inconsistency in precision (float<->double)

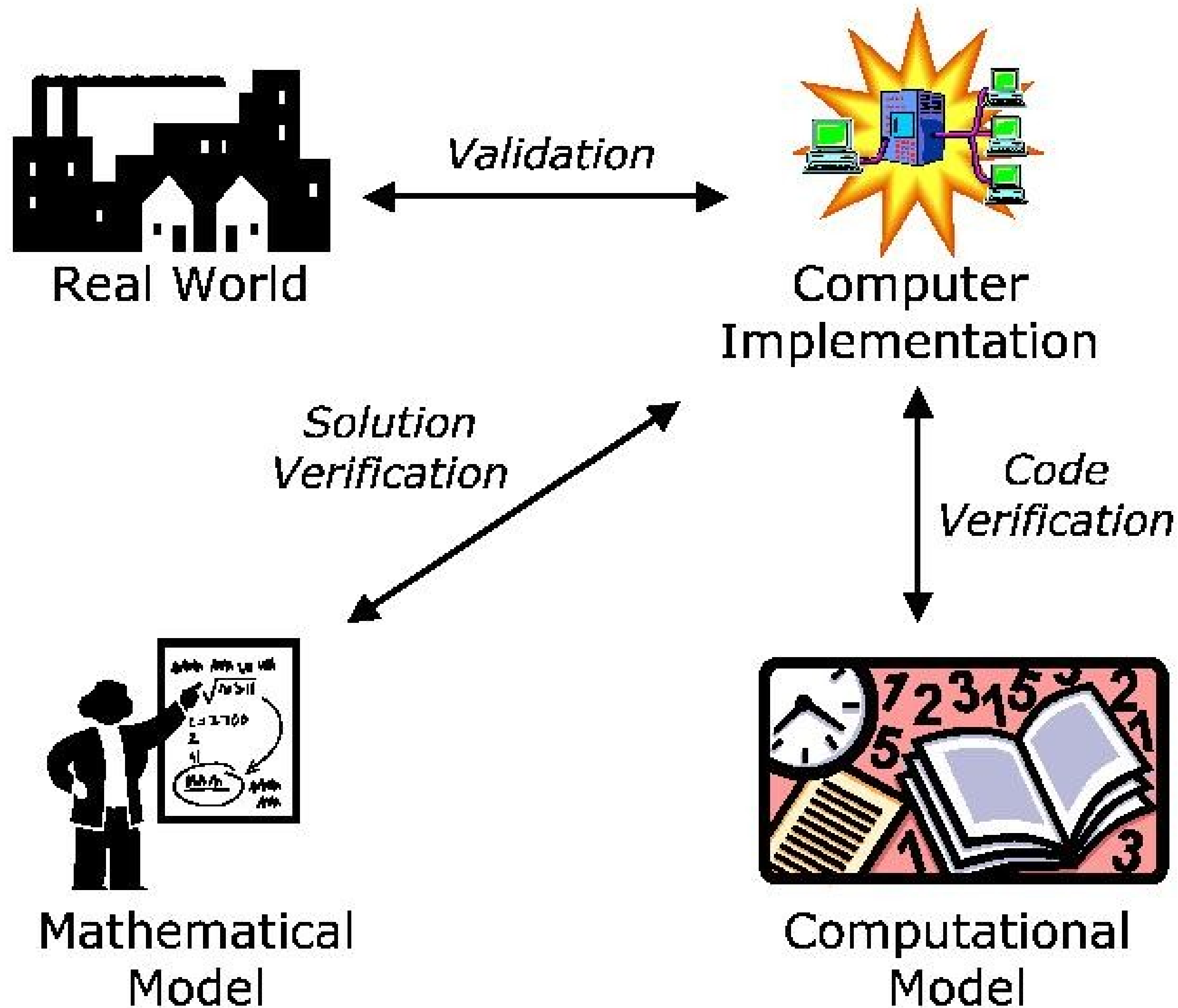
Erroneous stopping criterion

Wrong code producing likely results

Ill conditioned problem

Unstable algorithm, stability domain

Validation



Code verification

Static analysis: “dead code”, infinite loops, typing mistakes, bad type of variable

Dynamic analysis: debugger, profiler, printf

Tests:
Generate exact solutions (inhomogeneous problems)
Test modules separately
Choose extreme values for parameters (eg Navier-Stokes infinite viscosity)
Tests repositories (eg matrices)

Arithmetic error checking

Interval arithmetic

Allow to make proofs, eg convergence of a Newton method, singularity of a matrix

Require a specific coding

Do not allow to check a numerical code

Perturbations

Allow to check a software, to analyse the conditioning, to estimate the effects of data or algorithm errors

PRECISE: PRecision Estimation and Control In Scientific and Engineering computing

Solution verification

Test a problem with solution in the approximation space

Elliptic problem: convergence rate $h \to 0$

Convergence plot at a particular point

Solution for the finest mesh considered as being exact

Parabolic problem: steady state $T \to \infty$

Nonlinear problem: stricter stopping criterion

Detects instabilities

Problem on computer

Arithmetic effects can not be completely removed, but possibly decreased

Numerically ill-posed problems

$$y'(t)=10\mathrm{y}-10\mathrm{t},\ y(0)=y_0$$
$$\rightarrow y(t)=t+\frac{1}{10}+(y_0-\frac{1}{10})e^{10\mathrm{t}}$$
$$y_0=0.1:y(3)=3.1$$
$$y_0=0.0999999:y(3)=3.1-10^{-7}e^{30}=3.1-1.1*10^{6}$$

A problem is numerically well-posed if
1) It has a unique solution
2) This solution is Lipschitz-continuous with respect to the data with a small constant

Condition number

Measures the stability of the solution with respect to the data. **Depends only on the problem.**
Example: function evaluation

$$x = g(y), \quad \hat{x} = g(y+\epsilon)$$
$$g \in C^2 : g(y+\epsilon) = g(y) + \epsilon g'(y) + O(\epsilon^2)$$
$$\frac{x-\hat{x}}{x} = \frac{g'(y)}{g(y)}\epsilon + O(\epsilon^2)$$
$$\left|\frac{x-\hat{x}}{x}\right| \approx \left|\frac{g'(y)}{g(y)}\right| \epsilon = \kappa(g)\epsilon$$

Example: linear system

$$Ax = b \quad (A+E)\hat{x} = b+e \quad \|e\| \le \epsilon \|b\| \quad \|E\| \le \epsilon \|A\|$$
$$\hat{x} = (A+E)^{-1}(b+e) = (I - EA^{-1})A^{-1}(b+e) + O(\epsilon^2) = A^{-1}b + A^{-1}e - A^{-1}EA^{-1}b + O(\epsilon^2)$$
$$\hat{x} - x = -A^{-1}Ex + A^{-1}e$$
$$\|\hat{x} - x\| \le \|A^{-1}\| \left(\|E\| \ \|x\| + \|b\| \ \frac{\|e\|}{\|b\|} \right) \le \|A^{-1}\| \ \|A\| \left(\frac{\|E\|}{\|A\|} + \frac{\|e\|}{\|b\|} \right) \|x\|$$
$$\Rightarrow \kappa(A) = \|A^{-1}\| \ \|A\|$$

Stability

Measures the sensitivity of **the solution of a numerical approximation** with respect to the data:

$$y_n = \frac{1}{e}\int_0^1 x^n e^x dx$$

$$y_n = 1 - n y_{n-1},\ y_0 = 1 - 1/e$$

$$\widehat{y_1} = 1 - \widehat{y_0} = y_1 - \epsilon$$

$$\widehat{y_2} = 1 - 2\,\widehat{y_1} = y_2 + 2\epsilon$$

$$\widehat{y_3} = 1 - 3\,\widehat{y_2} = y_3 - 6\epsilon$$

$$\Rightarrow \widehat{y_n} = y_n + (-1)^n n!\,\epsilon$$

The method is unstable!

But the recurrence

$$y_{n-1} = (1 - y_n)/n$$

is stable!

Approximations hierarchy
(F. Chaitin-Chatelin)

Continuous problem

$$(P)\ \ F(x)=y\,,\ \ F: X\rightarrow Y$$
$$F\,,\ \ F^{-1}\,continuous\Rightarrow x=F^{-1}(y)$$

Numerical approximation (discretization, iterations and stopping criterion)

$$(P_h)\ \ F_h(x_h)=y_h\,,\ \ F_h: X_h\subset X\rightarrow Y_h\subset Y$$
$$\Leftrightarrow x_h=G_h(y_h)$$

Realization of the algorithm on a machine with precision ε

$$(\hat{P}_{h\epsilon})\ \ \hat{F}_{h\epsilon}(\hat{x}_{h\epsilon})=\hat{y}_{h\epsilon}\,,\ \ \hat{F}_{h\epsilon}: IF^m\rightarrow IF^n$$
$$\Leftrightarrow \hat{x}_{h\epsilon}=\hat{G}_{h\epsilon}(\hat{x}_{h\epsilon})$$

Examples

$$(P)\ \ x=\cos(y)\,,\ \ \cancel{(P_h)}\,,\ \ (\hat{P}_\epsilon)\ \ \hat{x}_\epsilon=\widehat{\cos}(\hat{y}_\epsilon):\ \ (P)\sim(\hat{P}_\epsilon)\,?$$
$$\cancel{(P)}\,,\ \ (P_h)\ \ Ax_h=y_h\,,\ \ (\hat{P}_{h\epsilon})\ \ \hat{A}\,\hat{x}_{h\epsilon}=\hat{y}_{h\epsilon}:\ \ (P_h)\sim(\hat{P}_{h\epsilon})\,?$$

Lax principle

Error of the discretized problem:

Consistency

$$F_h(z) \to F(z),\ \|z-x\| < \delta,\ y_h \to y,\ h \to 0$$

Uniform stability

$$h-\mathit{Equicontinuity}\quad \mathit{of}\quad G_h = F_h^{-1}$$

=> Convergence

$$x_h \to x,\ h \to 0$$

Example: Conforming finite elements

$$-\Delta u = f,\ \Omega,\ u = 0,\ \partial\Omega$$
$$X = H^1{}_0(\Omega),\ Y = L^2(\Omega),\ a(u,v) = \int_\Omega \nabla u \nabla v$$
$$a(u,v) = (F(u),v)\ \ \forall v \in X$$
$$X_h \subset X,\ Y_h = X_h$$
$$a(u_h,v_h) = (F_h(u_h),v_h)\ \ \forall v_h \in X_h$$
$$F_h \subset L(X_h)\ \ F_h(\phi_j)_i = a(\phi_j,\phi_i)$$
$$(F_h(u_h),v_h) = (P_h f,v_h)\ \ \forall v_h \in X_h$$

Analysis of rounding errors

Comparison of the discretized problem with it's computer version

$$(P_h) \ F_h(x_h) = y_h, \ x_h \in \mathbb{R}^m, \ y_h \in \mathbb{R}^n \Leftrightarrow x_h = G_h(y_h)$$

$$(\hat{P}_{h\epsilon}) \ \hat{F}_{h\epsilon}(\hat{x}_{h\epsilon}) = \hat{y}_{h\epsilon}, \ \hat{x}_{h\epsilon} \in IF^m, \ \hat{y}_{h\epsilon} \in IF^n \Leftrightarrow \hat{x}_{h\epsilon} = \hat{G}_{h\epsilon}(\hat{x}_{h\epsilon})$$

“Embedding” of the computer problem into real numbers

$$(P_{h\epsilon}) \ F_{h\epsilon}(x_{h\epsilon}) = y_{h\epsilon}, \ x_{h\epsilon} \in \mathbb{R}^m, \ y_{h\epsilon} \in \mathbb{R}^n \Leftrightarrow x_{h\epsilon} = G_{h\epsilon}(y_{h\epsilon})$$

Generally $F_{h\varepsilon}$ is not explicitly known!

Forward analysis: $x_{h\varepsilon}-x_h$ by comparing G_h to $G_{h\varepsilon}$

Backward analysis: $x_{h\varepsilon}-x_h$ by comparing F_h to $F_{h\varepsilon}$ using the Lax principle for ε!

Computability, reliability, stability

x is **computable** if x_ε goes to x when ε goes to zero

The algorithm defined by the solution of the computer problem is **arithmetically reliable** if:

$$\|G_\epsilon(y_\epsilon) - G(y_\epsilon)\| \le C(y, G)\epsilon$$

Usually, we can not get better than ε^1

Stability:

$$x = F^{-1}(y) = G(y), \; G(y + \delta z) = G(y) + DG(y)\delta z + O(\delta^2)$$

$$Conditioning: \|DG(y)\|, \; \|G(y+\delta z) - G(y)\| \le \|DG(y)\| \; \|\delta z\|$$

Example

$$y = Ax: \; G(y) = A^{-1}y, \; DG(y) = A^{-1}$$

$$\|G(y+\delta z) - G(y)\| \le \|A^{-1}\| \; \|\delta z\| \; \frac{\|y\|}{\|y\|} \le \|A^{-1}\| \; \|A\| \; \|x\| \; \frac{\|\delta z\|}{\|y\|}$$

Error:

$$\left\|\frac{G(y+\delta z) - G(y)}{G(y)}\right\| \le \|A^{-1}\| \; \|A\| \; \frac{\|\delta z\|}{\|y\|}$$

$$x - x_\epsilon = G(y) - G(y_\epsilon) \; + \; G(y_\epsilon) - G_\epsilon(y_\epsilon)$$

Stability of G + arithmetic reliability imply computability!

Wilkinson principle

Backward analysis: consider the solution of the computer problem as the solution of an exact perturbed problem

Example: linear system

$$Ax=b: \ Find\ E, e\ \ st\ \ (A+E)x_\epsilon = y_\epsilon = b+e,\ \|E\| \le c\|A\|,\ \|e\| \le c\|b\|$$

$$\eta_{A,b} = min\{c:\ (A+E)x_\epsilon = b+e,\ \ \|E\| \le c\|A\|,\ \|e\| \le c\|b\|\} = \frac{\|r\|}{\|A\|\|x_\epsilon\| + \|b\|},\ \ r = b - Ax_\epsilon$$

Existence of E and e?

Perturbations

Normwise, componentwise (sparse)

Example: linear system

$$\|E\| \le c\|A\|,\ \|e\| \le c\|b\|,\ (A+E)x_\epsilon = b+e$$

$$\frac{\|x-x_\epsilon\|}{\|x\|} \le \frac{c}{1-c\|A^{-1}\|\|A\|}\left(\frac{\|A^{-1}\|\|b\|}{\|x\|}+\|A^{-1}\|\|A\|\right)$$

$$\kappa(A,b,x) = \frac{\|A^{-1}\|\|b\|}{\|x\|}+\|A^{-1}\|\|A\| \le 2\kappa(A) = \|A^{-1}\|\|A\|$$

$$\frac{\|x-x_\epsilon\|}{\|x\|} \le 2\kappa(A)\frac{\|r\|}{\|A\|\|x_\epsilon\|+\|b\|}$$

Analysis on the computer

Problem y=F(x), approximate solution $x_\theta = G_\theta(y)$

$$y \xrightarrow{G_\theta} x_\theta \xrightarrow{F_\theta} \zeta = F_\theta(x_\theta)$$

Choose the perturbation model

Which data, which metric, which interval?

$$t = \|\Delta z\|: \;\; z + \Delta z \xrightarrow{G_\theta} x_\theta + \Delta x_\theta \xrightarrow{F_\theta} \zeta_\theta + \Delta \zeta_\theta = y + \Delta w_\theta$$

Generate indicators:

- Numerical stability $$I_{\theta t} = \sup_{\|\Delta z\| = t} \frac{\|\Delta w_\theta\|}{t}$$
- Stability and conditioning $$K_{\theta t} = \sup_{\|\Delta z\| = t} \frac{\|\Delta x_\theta\|}{\|\Delta \zeta_\theta\|}$$
- Algorithm sensitivity $$L_{\theta t} = \sup_{\|\Delta z\| = t} \frac{\|\Delta x_\theta\|}{t}$$

PRECISE

- Solution of linear systems

$$Ax=b \quad x_\theta=G_\theta(b) \quad F(x)=Ax \quad \zeta_\theta=F_\theta(x_\theta)=A_\theta x_\theta$$

- t fixed: perturbations sample

$$z_t=z+\Delta z=\begin{Bmatrix} A+E \\ b+e \end{Bmatrix} \xrightarrow{G_\theta} x_\theta+\Delta x_\theta \xrightarrow{F_\theta} \zeta_\theta+\Delta\zeta_\theta=y+\Delta w_\theta$$

$$\downarrow Z_t \qquad \downarrow X_t \qquad \downarrow Y_t$$

- Sample $\mathbf{X}=(X_1,X_2,\ldots,X_n)$

  Considered as observations of independent variables with the same distribution

- Mean $\mu=E(X)$, variance $\sigma^2=E((X-E(X))^2)$

  standard deviation σ

- Estimate μ(X) and σ(X) using the sample

$$m_n(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i = \overline{x} \qquad s_n^2(\boldsymbol{x}) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

- Unbiased estimates:

$$E(m_n(\boldsymbol{X})) = \mu(X) \qquad E(s_n^2(\boldsymbol{X})) = \sigma^2(X)$$

- Indicators: use σ for the reliability

$$L_{\theta t} = \frac{\|\sigma(X_t)\|}{t\|x_\theta\|}$$

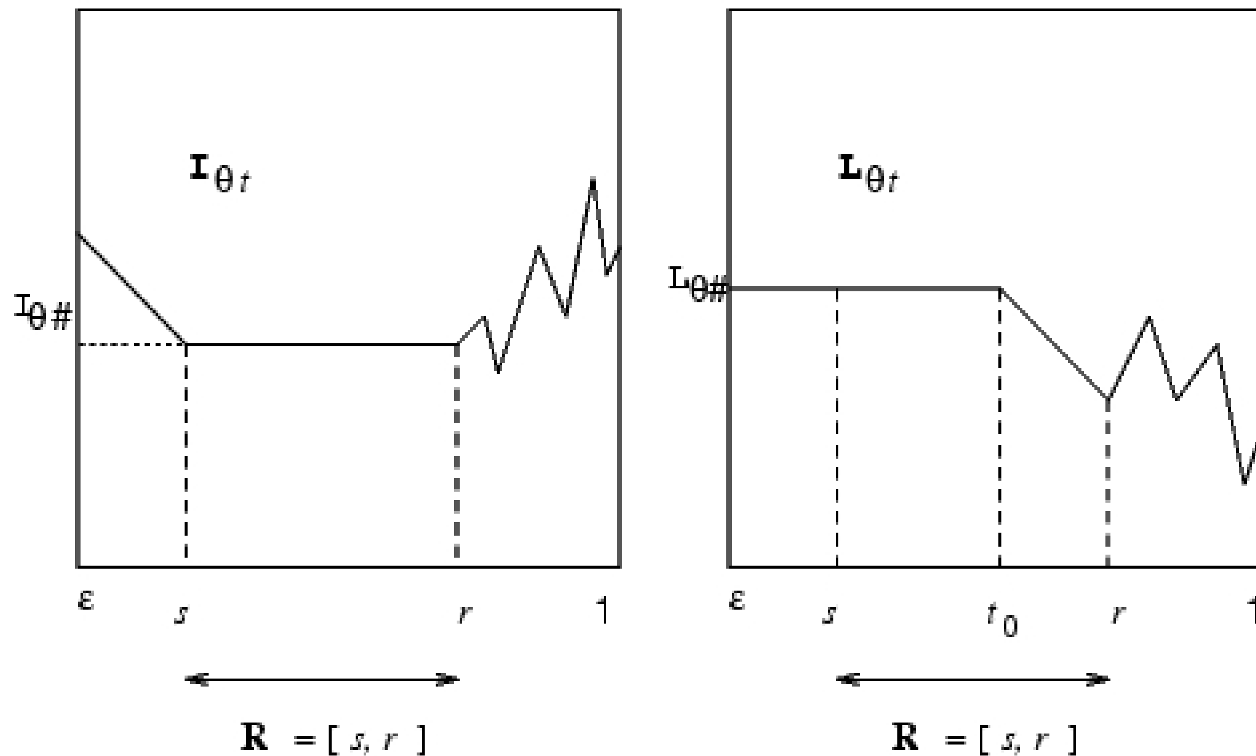
- Use also the backward estimate

$$K_{\theta t} = \frac{\|\sigma(X_t)\|}{\|x_\theta\|}\,\frac{\|\Delta A\|\|x_\theta\| + \|\Delta b\|}{\|\sigma(Y_t)\|}$$

- Standard deviation with respect to b

$$\hat{\sigma}_i(Y_t) = \sqrt{\sigma_i(Y_t)^2 + \left(\mu_i(Y_t) - b_i\right)^2} \qquad I_{\theta t} = \frac{1}{t}\,\frac{\|\hat{\sigma}(Y_t)\|}{\|\Delta A\|\|x_\theta\| + \|\Delta b\|}$$

- Typical behavior



- $I_{\theta t}$ *must be constant on an interval*

- *Estimate of the order of the backward error given by* s

Optimization for one CPU (core)

Parallelization = Vectorization

Computer in the 80's = PC now

Hardware means:

Pipeline (operations decomposition)

Multiples registers (+,*,...) working in parallel

Operations chaining

Vector registers and operations

Software means:

Compiler directives

Automatic compilers (for ex. Intel)

Vector coding

Vector coding: examples

Loop without dependency

```
for (i=0; i<n; i++) y[i]=x[i]+1;
```

Unrolling: 2 pairs enter into the pipeline

```
for (i=0; i<n; i+=2)
{ y[i]=x[i]+1; y[i+1]=x[i+1]+1;}
```

Pre-fetching: 1 variable enter before the next index

```
s = x[0];
for (i=0; i<n; i++)
{ y[i]=s+1; s=x[i+1];}
```

Unrolling of depth k + Pre-fetching

Automatic vectorization (ex: Intel compiler)

```
icc -xW -O3 -vec_report3 loop.c
```

Loop with dependency

```
for (i=0; i<n; i++) y[i]=y[i-2];
```

Can not be vectorized

Force vectorization: only the 2 first components are correct

Loop permutation

Matrix-vector product Ax=b by rows

```
for (j=0; j<n;j++)
    for(i=0; i<n; i++) b[i]=b[i]+x[j]*a[i][j];
```

Internal loop on the rows of A; the lines are stored consecutively

Matrix-vector product Ax=b by lines

```
for (i=0; i<n;i++)
   for(j=0; j<n; j++) b[i]=b[i]+a[i][j]*x[j];
```

Internal loop on the lines of A: is vectorized

Dot product

```
for (i=0; i<n;i++)
    b[i]=ddot_(&n,&(a[i][0]),&inc,x,&inc);
```

one uses the optimized function of the library BLAS

Computers: rough overview

Standard computer:

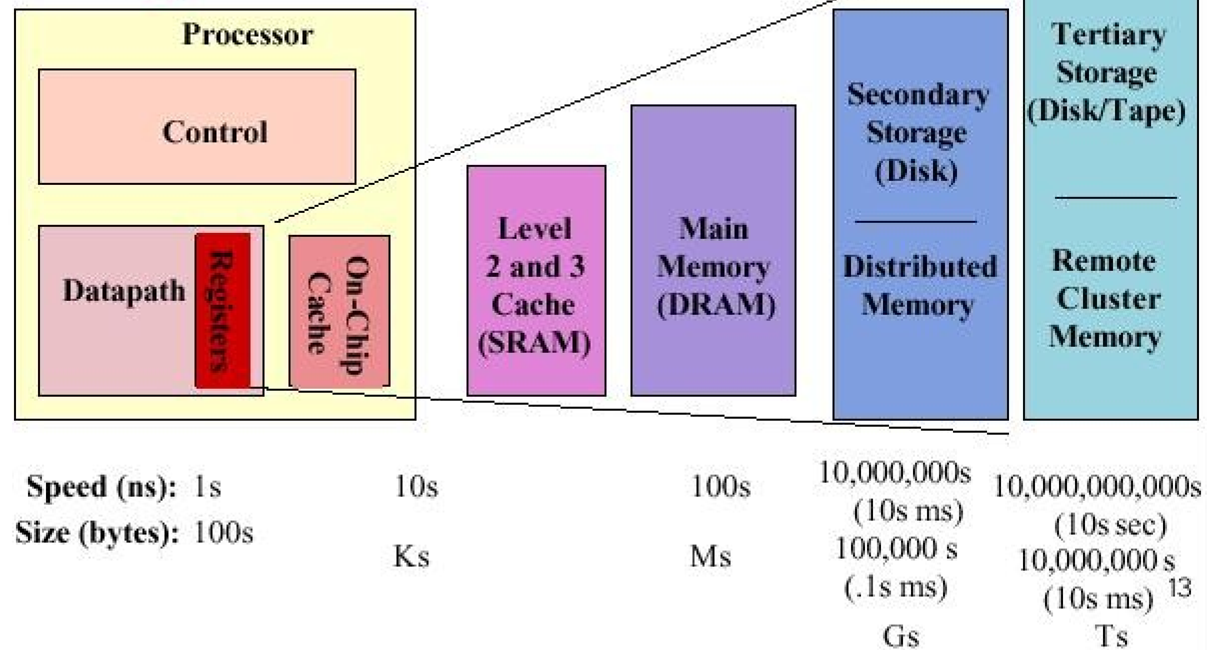
CPU (Central Processing Unit)
Main memory
Input-Output (I/O)

Scientific computing program:

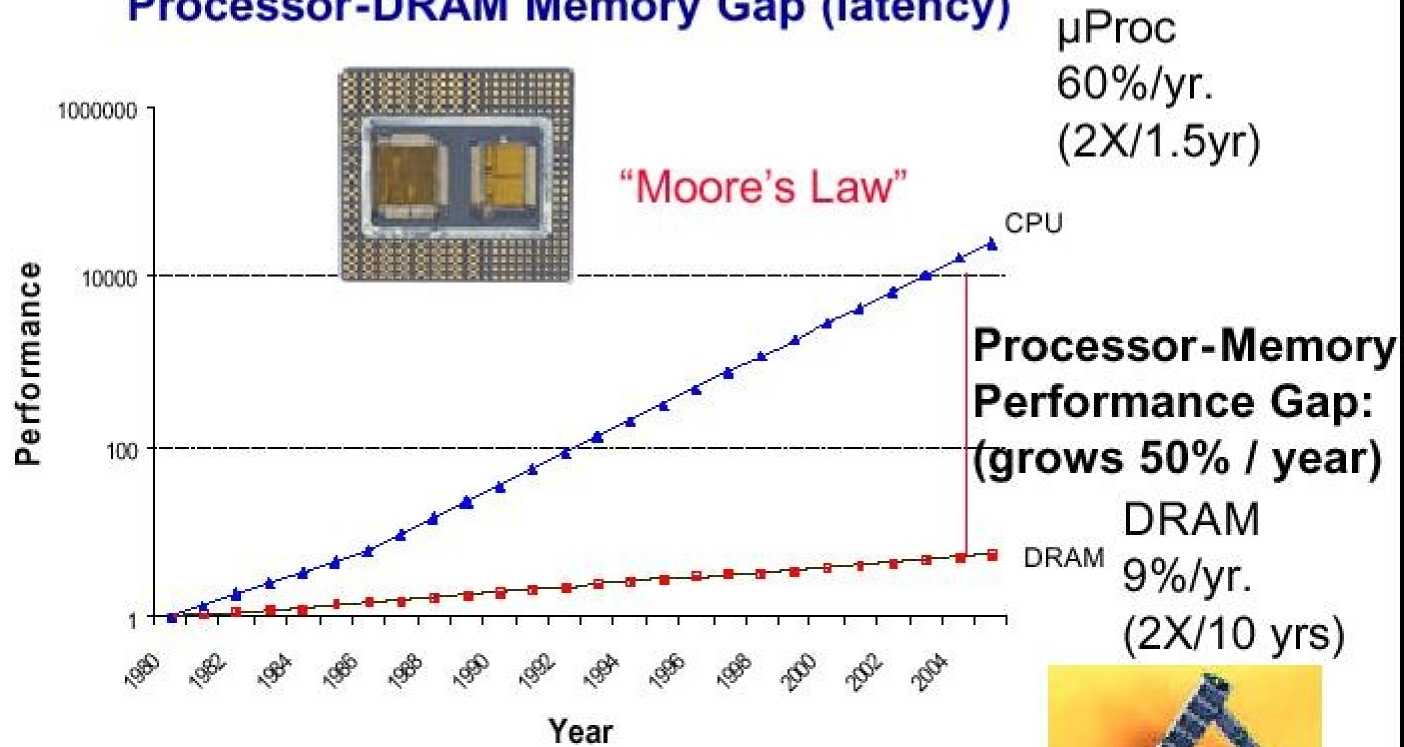
A lot of operations in the CPU
Transfer of a lot of data between the memory and the CPU

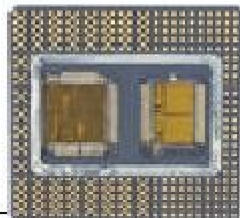
Memory Hierarchy

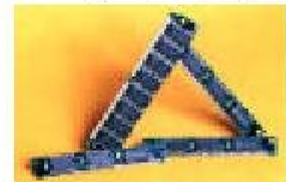
- **By taking advantage of the principle of locality:**
  - **Present the user with as much memory as is available in the cheapest technology.**
  - **Provide access at the speed offered by the fastest technology.**



Where Does Much of Our Lost Performance Go? or Why Should I Care About the Memory Hierarchy?







2 types of multi-CPU's computers

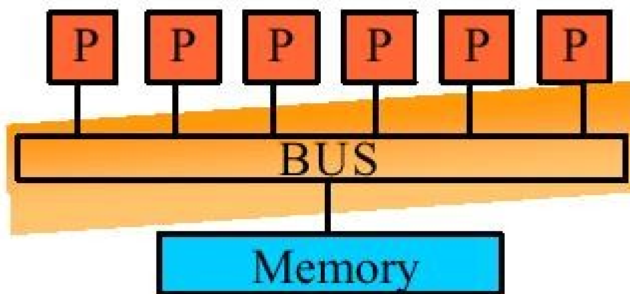
Memory model:

Shared memory
All processors have access to the whole memory
Data transfer management not necessary
(however: conflict when simultaneous access)

Distributed memory
Every processor can access only it's own local memory
=> Manage data transfer between processors

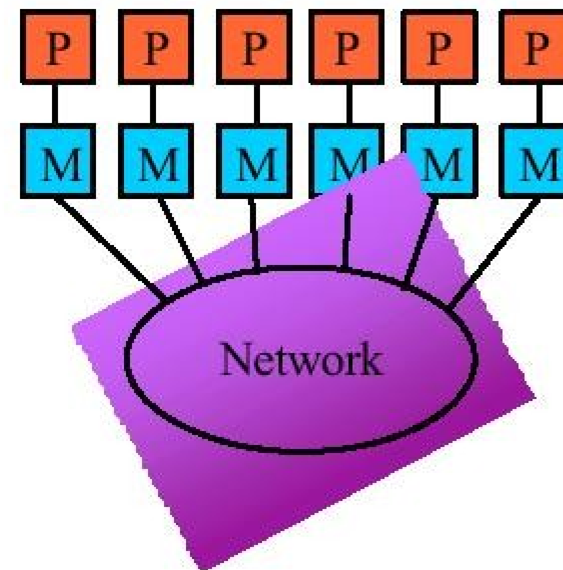
Mixed: Distributed memory, multicore CPU's

Shared vs. Distributed Memory



Shared memory - single address space. All processors have access to a pool of shared memory. (Ex: SGI Origin, Sun E10000)

Distributed memory - each processor has it's own local memory. Must do message passing to exchange data between processors. (Ex: CRAY T3E, IBM SP, clusters)



Distributed memory computers

Massively parallel computer (MPP)
1 computer, 1 OS, fast internal connections

Beowulf: N>124 CPU's, 1 OS, connections by fast network

Cluster: 1<n<124 machines, 1-n OS, connection by Ethernet; for example 10 PC's

2009: cluster = 2<n<... multicores CPU's with fast connection (myrinet, infiniband)

Better performance

Use fast memory
 That is: the one of each processor

Distribute computations between processors
 Decompose the problem

Equilibrate the amount of work of processors
 Load balancing

Example: Domain decomposition

PERFORMANCE (n cores)
Vocabulary (units)

Clock frequency:
Ghz (Pentium4 1-3)
number of floating point operations per second:
Flops
Mflops 10^6 Flops
Gflops 10^9 Flops
Tflops 10^{12} Flops
Pflops 10^{15} Flops
Pentium4 1-2 Gflops :
1 operation per ns

Top 500 (Cray XT 224162 cores):
2.33 Pflops

Total CPU time T(n) s

Speedup :
$S(n)=T(1)/T(n)$
Ideally: $S(n)=n$

Parallel efficiency:
$E(n)=T(1)/nT(n)$

Amdahl's law

<u>Speedup limited by the sequential part of the code</u>

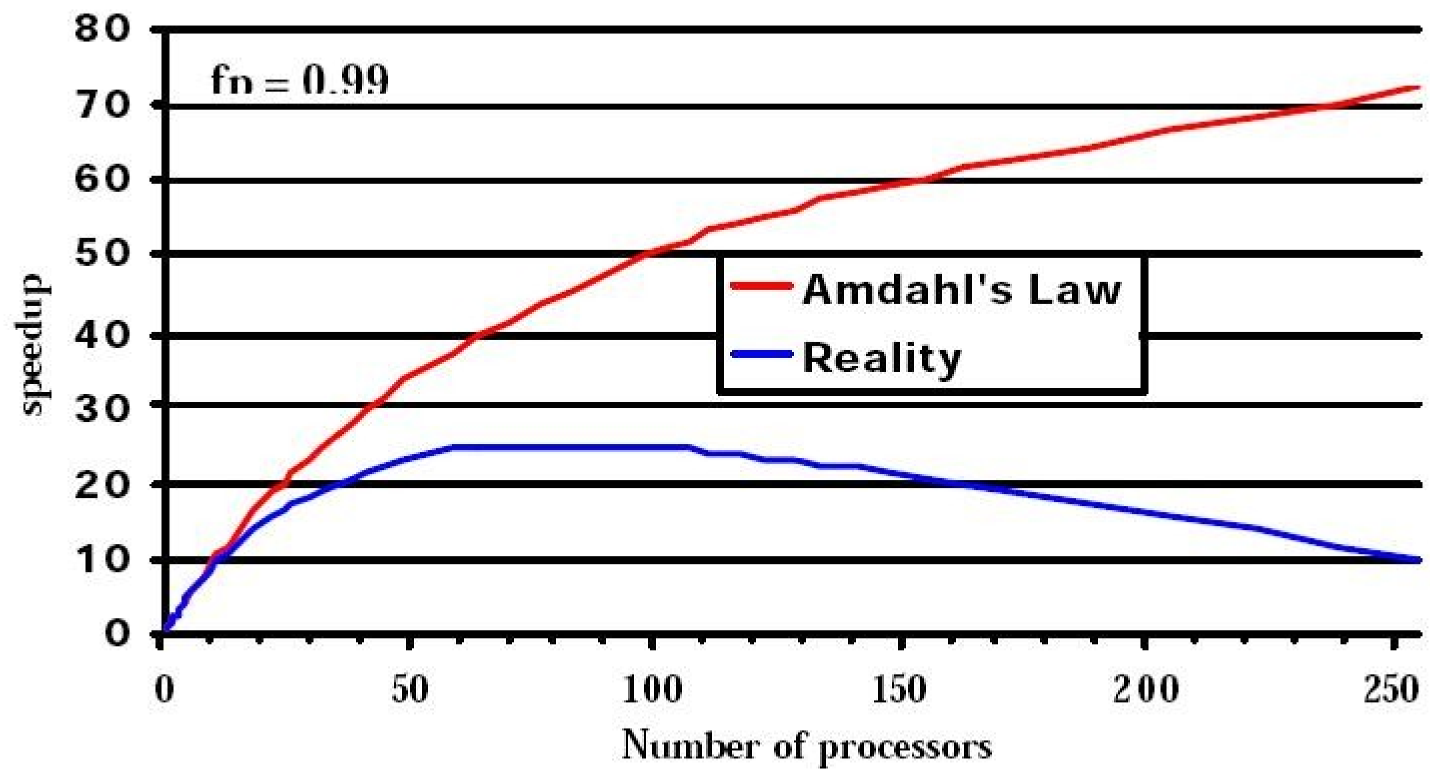
f_s: sequential fraction; f_p: parallel fraction =1-fs

$$T(n)=(f_p/n + f_s)T(1)$$

$$S(n)=1/(f_p/n + f_s)$$

The limit n tends to infinity depends only on f_s!

One can enhance the speedup by increasing the “size”


fp = 0.99
80
70
60
50
40
30
20
10
0
speedup
Amdahl's Law
Reality
0
50
100
150
200
250
Number of processors