Chapter 1

Introduction

1.1 Preliminaries

- Contact:
  - Antonius Otto
  - Geophysical Institute, 708C
  - Phone 474 6169
  - Email: ao@how.gi.alaska.edu
  - Web site: http://www.how.gi.alaska.edu/ao/sim/index.html
  - Office hours: Anytime
- Student names and background
- Encourage participation, criticism, and suggestions
- Scope of the course:
  - Introduce, analyze, and apply method's to solve PDE's.
    - background on PDE's
    - discrete representation
    - stability and accuracy
    - boundary and initial conditions
  - Aspects to consider:
    - efficiency
    - computer performance and architecture
  - Applications:
    - multi-body (particle) systems - examples

        * fluid systems:
            · steady state systems and equilibria
            · convection
            · instabilities

- Conduct (see also description above)
    - 1. Part (about 3/4): Methodology for convection and diffusion equations (simple systems to illustrate methods)
    - 2. Part(about 1/4): more specific problem and increasing complexity, e.g.,
        * simple equations and complex geometry
        * simple geometry and complex equations
    - Books (see description above)
    - Programming languages and tools: FORTRAN and IDL but other are acceptable
    - Midterm test (mid March?)
    - Grading
- Questions

1.2 Why Numerical Simulation

Traditional approaches:

- Experiment and observation
- Analytical theory

All **interesting problems** in physics, engineering, biology, and other sciences are **nonlinear and multi-dimensional.** Analytical solutions exist only in extremely rare cases. For instance:

- Classical gravitationally interacting three-body problem (has no analytical solution except for rather special cases).
- Weather prediction.
- Aerodynamics of airplanes, cars etc. (ironically this field of physics is one of the older without much progress in the nonlinear turbulent air flow).

Many of these problems are well posed in terms of governing equations and boundary conditions. -> Numerical Models!

Example: Advantages of numerical modeling in aircraft design:

- Reduced lead time in design and development
- Flow conditions are not available or are unrealistic due to scaling in experiments but are easily determined in simulation.
- In general: better information
- Reduction in energy consumption
- Reduction in overall cost
- Better products.

In many systems **observations or experiments** are **not available**, difficult, or and/or incomplete to describe the state. For instance

- Combustion problems
- Various categories of unsteady flow (changing geometry)
- Oil reservoirs
- Ground water flow
- Meteorology, weather and climate systems
- Solar and magnetospheric dynamics.

Science areas where numerical models are essential:

- Nonlinear dynamics, chaos, self-organized criticality
- Population dynamics
- Astrophysics (stellar dynamics, galaxy and star formation, cosmology)
- Plasma physics: Fusion studies, space physics
- Quantum mechanical reactions, high energy physics, particle decay and collisions, etc
- Nuclear and molecular structure
- Genome project and genetic engineering
- Neural networks

1.3 The Numerical Experiment: Limitations, Accuracy, Stability

1.3.1 Limitations

Physical/mathematical limitations: Physical (chemical ...) models must appropriately describe the system under study. For instance, the density of a gas, a liquid, or the concentration of dust in air (or any other medium) with a known velocity $\mathbf{u}$ is determined by

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot \rho \mathbf{u}$$

if mass is conserved. Here $\rho =$ mass density. Note also that $\nabla \cdot \mathbf{u} = 0$ implies $\partial \rho / \partial t = -\mathbf{u} \cdot \nabla \rho$ or $d\rho/dt = 0$ where d/dt is the total time derivative along the fluid trajectory.

Self-consistency requires that $\mathbf{u}$ is determined by another equation for the velocity of the medium:

$$\frac{\partial \rho \mathbf{u}}{\partial t} = -\nabla \cdot (\rho \mathbf{u}\mathbf{u} + \underline{\underline{1}} p + \underline{\underline{\mathbf{w}}}) + \mathbf{f}$$

with an isotropic pressure p, a viscosity $\underline{\underline{\mathbf{w}}}$, and a force density $\mathbf{f}$. For instance in the case of a magnetofluid this force can be the $\mathbf{j} \times \mathbf{B}$ force or in the case of gravity $\mathbf{f} = \rho \mathbf{g}$. Additional equations may be required for the pressure, viscosity, etc. Further, the actual system can consist of multiple sets of these equations with source terms for mass (generation and loss due to chemical reactions), momentum (friction between different constituents), and energy. Clearly the physical description of the system by the chosen equations must be accurate for the considered problem (e.g., length and time scales).

Limitations due to computer resources: Another set of limitations arises through constraints regarding the physical size of the problem and limited computer resources like memory.

Particle simulation of air: The number density of air under regular atmospheric pressure is $2.6 \cdot 10^{25}$ particles per m^3. Modern supercomputers have memory of about 10^{14} words$\sim 10^{15} Bytes =$ 1000 Terabytes (1 word = 1real number; for 100,000 nodes or about a million processors) to store coordinates and velocities of all air particle in 1 m^3 such a computer is short of memory by a factor of about 10^{12}. The volume that could just about be simulated on this machine (tracing all air molecules) is $(0.1\mathrm{mm})^3$.

Weather simulation: Assuming the size of a simulation box of $(100\mathrm{km})^3$ (this may be a little high in altitude but also a bit low on area) with the goal to resolve air turbulence (which has a length scale of about 10 cm) requires a grid of about 10^{18}. Thus modern computers are short by more than a factor of about 10^5 to resolve the fine scale turbulence in a weather simulation.

Space plasma simulation: The density of charged particles in space is approximately 1 per cm^3. Typical dimensions for dynamics space plasma processes (in the magnetosphere) are a few

Earth radii. Considering a box with a base length of 20,000 km($\approx 3R_E$), this box would contain 10^{28} charged particles. This requires to improve computer memory by a factor of 10^{15} to carry out this simulation with the real particle density. Actual box that can besimulated is $(20km)^3$.

While these examples may not sound encouraging, the situation is better than illustrated. For instance, in many cases the main effect of small scale turbulence is enhanced (turbulent) viscosity and diffusion. These effects can be parametrized for many applications such that it is not needed to resolve the actual fine scale turbulence. Similarly, there are ways to reduce the information due to the large number of particles. For a large number of problems the discreteness of the particle dynamics does not matter. In those cases fluid approximations of these systems may yield excellent results for the relevant parameter range. In cases where the particle dynamics is important, one can often re-scale charge and mass of the particles (in other words simulate super-massive particles) and yield the desired effects. It is important in all of these cases, to keep the approximations (physical and numerical) in mind and to make sure that the resulting errors can be neglected for instance through careful testing and methodology.

1.3.2 Accuracy

The discretization always introduces errors and there are various types of such error sources. Typical errors for a discretization are due to

- Truncation and rounding.
- The discrete representation and the limited computer resources (memory and speed).
- Accumulation of errors in a systematic manner for many numerical methods.

The way to improve on the results is to reduce the respective error source. This can be done for instance by:

- Better spatial and/or temporal resolution (smaller grid space, more finite elements, larger number of base functions, smaller time step).
- An increase of the number of particles.
- A change to a better and more appropriate method.

But deterministic chaos: A deterministic system is a system which is exactly predictable, i.e., there is an equation which describes the evolution of the system for all times for any exact set of initial condition. Specifically, this also implies that the set of equations is temporally reversible. Example: a ball on a plain surface with constant slope. However, more complicated systems frequently exhibit a property called deterministic chaos, that is, for any small deviation ε from a specific initial condition the system is arbitrarily far from the correct state after a finite amount of time. Example: In pool billiard a sufficiently accurate determination of the position and momentum of the white ball after 7 collisions requires to determine the initial conditions better than Heisenberg's uncertainty principle allows.

1.3.3 Stability

A particularly poor error evolution is caused by numerical instabilities. The discreteness of the numerical algorithm always introduces modifications to the equations which are to be solved. These can be expressed for instance in a dispersion relation. The errors which are introduced have two dominant effects. They change the dispersion (phase speeds) properties in particular for short wavelengths (measured in units of the grid separation). The errors also alter diffusion and if the effective diffusion becomes negative the errors grow exponentially resulting in a numerical instability.

Another way to demonstrate such behavior is the following. Explicit finite difference methods for hyperbolic equations are subject to a stability limit for the time step

$$\Delta t \leq O(\Delta x / v_{typ})$$

known as the Courant condition (or the CFL=Courant-Friederichs-Levi condition). Explicit differencing implies that the update is obtained with quantities known in the vicinity of the particular node i, i.e., usually requires nodes $i-1$, i, and $i+1$. Let us assume information is carried in the simulation with a velocity v_{typ} implying that this information is carried a distance $l = v_{typ} \cdot \Delta t$ in a single time step Δt. This information could be a wave or a temperature front etc. and the typical velocity could be a phase velocity like the speed of sound or just a moving air stream. However, if l is much larger than the grid separation Δx than it travels more than a single grid spacing in one time step. Since the update of density or temperature (or similar) uses only local information, the arrival of a wave front cannot be predicted correctly at any given grid point or node. In other words, the maximum velocity that is resolved by an explicit scheme is given by $v_{max} = \Delta x / \Delta t$ yielding the condition $\Delta t \leq v_{typ} / \Delta x$ as in the Courant condition above. While this consideration does not not necessarily imply an instability it makes clear that a too large time step is not able to resolve the proper propagation of information in the system.

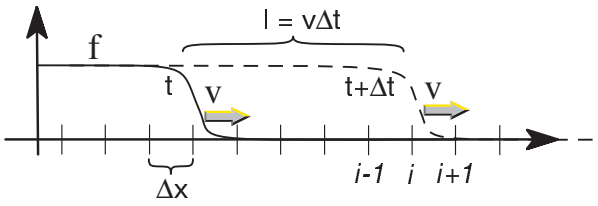

Figure 1.1: Illustration of a wave front traveling with a speed faster than v_{max} across a uniform grid.

1.4 Vector and parallel computer performance

1.4.1 Vector processing concept:

A vector processor can load an entire array of numbers or operate on such an array in a single cycle, different from a scalar processor which always loads or operates on single numbers (often

the width of the processor, e.g., 32 bit (= 4 byte real numbers).

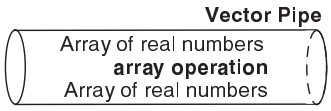

Figure 1.2: Illustration of the vector processing concept

Assuming

- a total number of operations $\equiv N$
- a vector speed (number of ops/sec in vector mode) $\equiv V(N)$
- a scalar speed $\equiv S$

we can determine the total execution time if all operations are scalar as $\tau_S = N/S$ and the total execution time if a fraction P of the problem can be vectorized:

$$\tau_V = \frac{N}{S}(1-P) + \frac{N}{V}P$$

The resulting speed-up between scalar and vector operation for a given problem

$$\frac{\tau_S}{\tau_V} = \left(1 - P + \frac{S}{V}P\right)^{-1} \tag{1.1}$$

is called Amdahl's law. Since the scalar speed is rather small (typically a few % of the vector speed) the fraction of a problem that is vectorizable must be very high to reach execution speed-ups close to the maximum of V/S. Note, that this treatment is idealized because scalar and vector speed may depend of the type of operations and the actual vector speed could also depend on the size of the problem.

Exercise: Plot the speed-up as a function of P for $V/S = 100$.

Exercise: What is the approximate speed-up if half of a code (no. of op's) is vectorized?

1.4.2 Parallel processing concept

A parallel processing supercomputer usually consists of many fast scalar processor which can be used to address a problem simultaneously with any number of processors available. A critical bottle neck for the execution of a program is access to memory. Memory can be shared between processors (that is processors have access to the same memory) or distributed among processors. In the latter case the logic of the execution of a larger program is more complicated because it requires to exchange data between processors. Particularly for large numbers of processors memory has to be distributed in some fashion because of limited bandwidth and resulting slow access if all

processors would share the same memory. Modern architecture often use a hybrid model with so-called nodes. Each node consists of some number of processors (16 or 32) among which memory is shared, however, among nodes memory is distributed. This way applications which are not written for distributed memory have still a parallelization advantage. In the following illustration of some aspects of parallelization we assume a simple model assuming that memory among processors is distributed.

Consider an assembly of processors which have local memory and which are linked with fast data connections. The number of data links per processor depends on the particular hardware and also determines in part the overall topology of how the processor are connected to each other (e.g., hypercube ..). The distributed memory requires inter-processor communication for almost all problems.

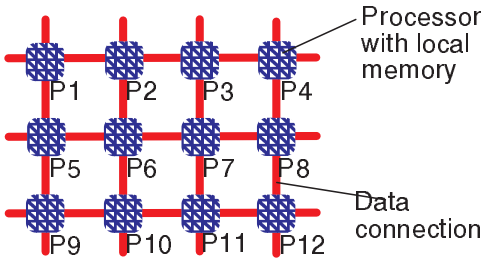

Figure 1.3: Illustration of the parallel processing concept

Parallel programming requires to distribute a given numerical problem to N_P processors in a manner that all processors have an approximately equal load.

Exercise: What would happen if they don't?

Since memory is usually distributed (on board of each processor.) one has to insure that all processors have the information they require to perform their individual portion of the problem. Defining

- $S \equiv$ speed of the individual processor
- $F \equiv$ efficiency of the parallelized code (dependent on parallelization method, number of processors, and problem size). For instance an unequal load for the processors may cause some processors to idle (F does not account for a situation where all processors except for one are idle. This is accounted for by $P < 1$)
- $N_p \equiv$ number of processors
- $P \equiv$ fraction of problem which can be parallelized
- $\tau_C \equiv$ time for inter-processor communication (problem dependent)

the total execution time for a problem of size N is

$$\tau_P = \frac{N}{S}(1-P) + \frac{N}{FSN_p}P + \tau_C$$

resulting in a speed-up of

$$\frac{\tau_S}{\tau_P} = \left(1 - P + \frac{P}{FN_p} + \frac{\tau_C S}{N}\right)^{-1} \tag{1.2}$$

for the parallel execution compared to a scalar execution. This is Amdahl's law for parallel processing with distributed memory. The maximum speed-up is N_P and factors reducing this speed-up are the achieved efficiency, the fraction of the problem that cannot be parallelized, and any considerable inter-processor communication time. Note again that the actual speed depends on the type of operations and properties such as F, P, and τ_c depend on the particular method implemented for the parallelization.

Domain decomposition*:*

One of the challenges of parallel processing is the distribution of a given numerical problem to many processors each of which dealing with a certain portion of the overall computation. There are many ways to distribute any given problem to multiple processors and actual method depends on the particular problem to be solved, the available software (operating system and compilers), and the hardware (i.e. how many processors are available, how are they communicating, and how is memory distributed).

For instance a long loop which executes an explicit matrix operation (i.e., an operation like a multiplication of a matrix with a vector where all matrix and vector elements are known) can just be cut into N_p parts, each of which is given to a particular processor. Many vector supercomputers have multiple vector processors and can automatically parallelize such an operation. This is possible without inter-processor communication because most vector supercomputers use a single central memory such that all information can be accessed very fast by each of the vector processors.

Parallel supercomputers, however, often have distributed memory, i.e., each node (or processor) has its own memory and any information which is not stored on the local processor requires communication with other processors in the machine. Some tasks in numerical modeling indeed require only local information while others may require information from neighboring nodes (in the case of finite differences, finite elements, etc) or from all nodes (global integral information). The following example illustrates some of the problems and inefficiencies which may be encountered in parallel processing.

Let us consider a three-dimensional domain which uses $N = n^3$ grid points (or finite elements, finite volume elements, or system of base functions) where n is the number of grid points in each direction as illustrated in Figure 1.4. Note that n is chosen to be the same in each direction for simplicity. The task is now to associate certain sub-domains (domains of grid points) with certain processors. There are three relatively simple ways to cut the three-dimensional domain into N_p pieces as illustrated in Figure 1.5.

In the first case the domain is sliced along one dimension only. Note that the figure only shows the x, y plane of the 3D system. In three dimensions this produces slices with the dimension $\frac{n}{N_p} \times n \times n$ for the x, y, and z directions. Here we assume n/N_p and all other integer operations in the following

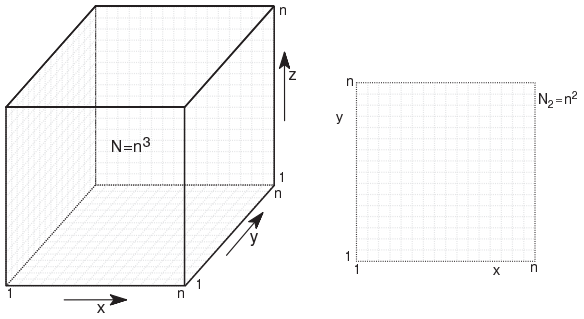

Figure 1.4: Illustration of a three-dimensional gridded system (left) and a cut in the x, y plane.

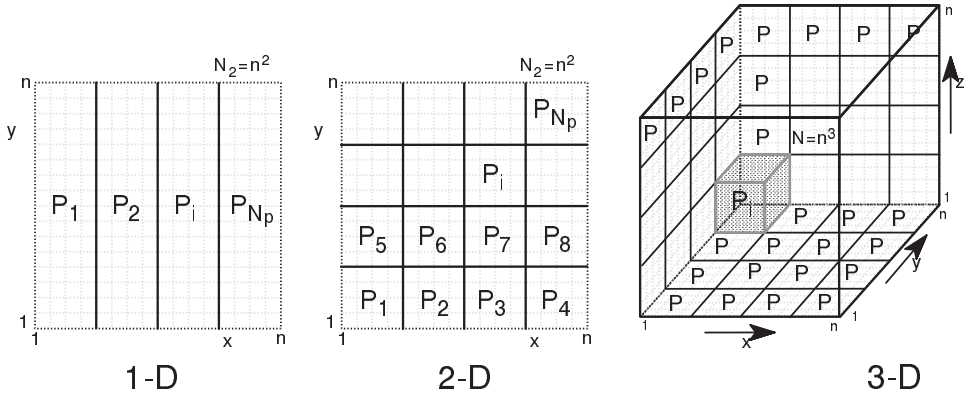

Figure 1.5: Illustration of three different (one-, two-, and three-dimensional) domain decompositions.

yield integer results for simplicity. The surface area with the neighboring domain is $2n \times n$ grid points for each (interior) processor.

In the second cases the domain is sliced in two directions yielding a two-dimensional decomposition. The 2-D sketch in Figure 1.5 shows the base of columns each of which has the dimension $\frac{n}{\sqrt{N_p}} \times \frac{n}{\sqrt{N_p}} \times n$ in the x, y, and z directions (assuming $\sqrt{N_p}$ and $n/\sqrt{N_p}$ to be integer numbers). The surface area with the neighboring domains is now $4\frac{n}{\sqrt{N_p}} \times n$ grid points for each (interior) processor. Finally, slicing the entire grid in three directions yields the situation illustrated in the 3-D sketch in Figure 1.5 where each processor works on a domain of $\frac{n}{\sqrt[3]{N_p}} \times \frac{n}{\sqrt[3]{N_p}} \times \frac{n}{\sqrt[3]{N_p}}$ grid points for x, y, and z. The surface for each sub-domain (processor) is $6\frac{n}{\sqrt[3]{N_p}} \times \frac{n}{\sqrt[3]{N_p}}$.

Let us consider that the computation at each particular grid point requires some information from the next neighbor point (for instance to determine a derivative from finite differences or finite elements). The elementary computation time for a single operation is $\tau = 1/S$ and and the elementary communication time for the exchange of the required next neighbor information between processors is τ_{ce} (i.e., for a grid point on the boundary of a sub-domain). If τ_{ce} is a constant fraction b of the elementary computation time ($\tau_{ce} = b\tau$) the total execution time for the entire problem is

$$\tau_P = N(1-P)\tau + \frac{N}{FN_p} P\tau + N_s b\tau$$

where N_s = number of grid points points on the surface of the sub-domains. With $n = \sqrt[3]{N}$ we obtain $N_s = 2dN^{2/3}N_p^{-\alpha}$ with $d = 1$, 2, and 3 for the 1-D, 2-D, and 3-D domain decomposition and and $\alpha = (d-1)/d$. Assuming perfect parallelization, i.e., $P \approx 1$ and $F \approx 1$ we obtain

$$\tau_P = \frac{N}{N_P}\tau + 2bd\frac{N^{2/3}}{N_p^{\alpha}}\tau. \tag{1.3}$$

Here the first term represents the total computation time and the second the total communication time. This demonstrates that the total execution time depends on the domain decomposition. A parallel execution becomes inefficient if a significant fraction of the execution time is spend on inter processor communication, i.e., when the second term in (1.3) equals the first term or when the ratio R of total communication time and total computation time is of order unity

$$R = 2bd\frac{N_p^{1-\alpha}}{N^{1/3}} = O(1). \tag{1.4}$$

R increases fast with the number of processors and depends strongly on the chosen domain decomposition. Thus it may not be beneficial to increase the number of processor beyond a certain level. The speed up gained compared to single processor execution time $\tau_S = N\tau$ is

$$\frac{\tau_S}{\tau_P} = \frac{N_p}{1+R} \tag{1.5}$$

Exercise: Derive equations (1.3), (1.4), and (1.5) with the assumptions made in the text.

Exercise: Plot execution time, R, and speed up as functions of N_p for the different domain decompositions.

1.4.3 Computer performance through the years

Numerical modeling and simulation has strongly benefited from a continuous increase in computer speed. There are various measures for the usefulness of computers, not all of which are hardware dependent. However for numerical modeling a central measure is the number of floating point operations per second. Note that this is not just a single fixed number for each hardware since different operations (add, subtract, multiply, and divide) usually take different amounts of time (depending on the implementation of these operations). Also the actually performance depends on the test suite (the programs used to determine execution time) and to what degree these have been optimized for a certain computer system. Therefore all the following numbers bear some uncertainty and usually do not reflect the peak performance but rather a performance which is actually achievable in real application (smaller than the peak performance by a factor of 2 to 10).

Table 1.1: Computer performance for different systems from 1984 to 2013. Numbers indicate performance in MFlops.

Year	PC-Type	Workstation	Mainframe	Vector SC	Parallel SC
1984	0.02 Intel 8086		3 Cyber 855	17 Cyber 205	
	0.037 Mot 68000				
	0.005 Atari ST				
1987	0.13 Intel 386	0.36 IBM RT		100 Cray XMP	
		0.51 Apollo			
1990-91	1 Intel 486	3 Sun Sparc 1		200 Cray YMP	
1993-94	7 Intel P5	20 Dec Alpha		450 Cray C90	
1997-98	40 Intel P6	100 Dec+others			$2*10^4$ T3E
2000-01	250 Pent 1 GH	500 SGI+others		1500 Cray SV1	$2*10^5$ T3E
2002-03	1000 Pent4 3 GH				10^6 Cray X1
2004-05	2k Opt. 2.4 GH				
	Pent 4 3 GH				
2006-07	5k Opt 2 cores				10^7Cray XT3
	Xeon 5160 2 co.				(12590 Opt)
2008-09	10k Xeon E5520	100k GPU Tesla			10^9 IBM Roadrunner
2010-11	30k Core i7 980	200k M2050			$2.5 \cdot 10^9$ Tianhe-1
2012-13	50k i7-3770	300k 2 E5-2470			10^{10} BlueGene/Q

This applies particularly to large parallel supercomputers where real applications often run by a factor of more than 10 slower than the peak performance or even the performance achieved by simple benchmark tests such as the linear algebra based LINPACK.

The first personal computers which were fairly universal in usage and which had a sufficient capacity to use for small numerical models were probably the Intel processor based PCs (together with other systems of about the same time like Motorola based PC's). Therefore the performance table 1.1 and corresponding Figure 1.6 will start with computers from the early to mid 1980's and go to the present.

The table gives an overview of the processor performance for PC type computer, workstations, vector supercomputers, and parallel super computers. Except for the parallel machines all performance numbers before 2008 reflect the speed of single processor systems (even though most vector computer had between 4 and 32 processors). Each entry gives the speed measured in so-called MFLOPS (million floating point operations per second) and the particular system line or processor name. The numbers quoted are often between a factor of 2 to 10 below the so-called peak performance.

Table 1.1 demonstrates the enormous progress in computer performance. For instance workstations and PCs of 2013 are more than an order of magnitude faster than vector supercomputers in the late 90's only a bit more than a decade ago and about as fast as parallel supercomputers from that time. These increase in computer speed is clearly demonstrated in Figure~11.6 which summarizes the data of the table on a logarithmic plot.

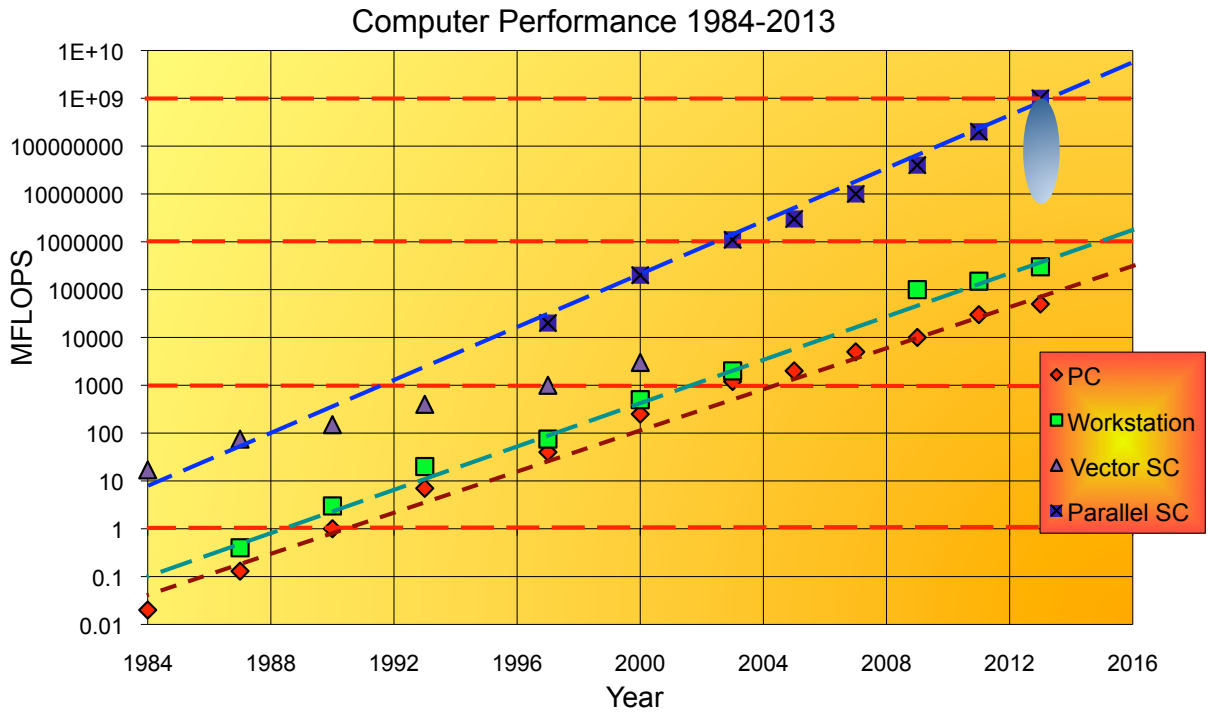

Figure 1.6: Computer performance throughout the years

The slope in Figure 1.6 for PCs demonstrate an increase in computer speed of a factor of 10 every 4 years (Any given point in Figure 1.6 and Table 1.1 may have an inaccuracy of a factor of 2 or so). A similar evaluation of computer memory and typical disk space would yield a slightly lesser increase in those resources. An amazing property of the computer evolution is the convergence of PC performance (and workstations) and that of vector supercomputers. Supercomputers offered a 1000fold advantage over PCs in the mid eighties, which is still the case only that massively parallel supercomputers have replaced vector supercomputers. Since these often use off the shelf PC or workstation microprocessors which can easily be upgraded the difference between massively parallel computers and PCs/workstations can be expected to remain at roughly a factor of 1000. However, this comparison is applicable only if the available parallel machine is state of the art. With a tenfold increase in PC performance over 4 years a typical 4 year old parallel supercomputer may in fact be of marginal advantage compared to the latest PC/workstation generation. Whether or not this increase in speed can be extrapolated further into the future is an open question. Figure 1.6 indicates a slightly faster increase of processor speed in the 80's and 90's compared to the last 15 years in PC's and workstations. There are also various limitations to the current technology which concern heat, radiation from extremely high cycle frequencies, and general quantum mechanical limitations regarding the size of elementary silicon based circuits.

In recent years all computers including PCs use multi-core processors where for the performance data 4 to 6 cores are assumed (e.g., Intel Core i7 980 and Xeon E5-2470). This means that since the introduction of multi-core PC processors (~2005) a significant part of the performance increase of 50 is due to the use of multiple cores and processors. A typical current supercomputer around 2010 is the CRAY XE6 (at the time at ARSC replaced by a smaller XK7) with 728 nodes and each node has two 8-core AMD processors for a total of 11648 computing cores. Compared to a fast

PC of that time, which can have one or two 6-core processors we again arrive at factor of about 1000 based on the number of processors in the supercomputer relative to the PC. While typical supercomputers in 2013 have about 1000 processors and 10000 computing cores, the fastest machines (top in 2012: Sequio - BlueGene/Q from IBM with $1.5 \cdot 10^6$ cores and a power consumption of 7.9 MW or 5 W/core) have 10^5 processors and about a million computing cores. The peak performance of these machines is about 10 PetaFlops or 10^{10} MFlops about 100 times faster than typical parallel supercomputers. This range of performance for supercomputers is indicated by the shaded oval in 1.6. The apparent small divergent between the slopes for PC's and Supercomputers applies only if one considers the top performing supercomputers and it disappears for typical parallel supercomputers.

Note that the time of dedicated processors for workstations (for scientific work) ran out in the early 2000's because PC processors (AMD and Intel) made faster progress than the more powerful but more specialized workstation processors. Today, gaming and entertainment applications are the dominant driving force for faster PC performance. Therefore processors for scientific work are usually just the top end of the mass produced PC processors. There is one exception which are computers based on GPU's (graphics processors). Typical graphics cards have considerably more raw computing power than CPU's because they employ large numbers of small computing units. These have been employed to produce very powerful small computers based on GPU's. However, since these often use hundreds (up to ~1000) of computing units with typically rather limited memory, it can take considerable effort to develop simulation models that can take advantage of GPU computing.

Table 1.2: Execution time for different computers and the problem $7 * 13$. *The time for the Cray XE6 includes time for an user application on ARSC supercomputers (It used to be weeks when there was a security check!

Computer	Time
Human	3 seconds
Pocket Calculator	30 seconds
PC	5 minutes
Cray XE6	days

There are further aspects to consider regarding computer performance. Problem size should match the potential computer performance. To illustrate tho point consider the problem $7 * 13$. The results are summarized in table 1.2. A related aspect is the time to develop a computer model which is considerable for parallel computers with distributed memory. The distributed memory requires instructions on how information is passed between processors. Automatic parallelization is possible only for very few simple problems and is usually not very efficient. Similarly all performance evaluations depend strongly on the particular application. Execution speed depends on fast memory access and sufficiently large memory and cache. Sufficient data storage or fast graphics can be other important factors. In the same category computer processors may show large variation in speed depending on the particular operations (i.e., integer, assignment, floating point operations etc.) used in a program. Finally all performance evaluation is carried out under the premise that the computed results are meaningful and measures of accuracy or comparison with

observations are applied to achieve confidence in the results. If this is not the case the performance is zero.