

Computational Astrophysics I: Introduction and basic concepts

Helge Todt

Astrophysics
Institute of Physics and Astronomy
University of Potsdam

SoSe 2024, 20.6.2024



MC Error estimation

Numerical integration (exact or MC) gives approximation

$$\int_a^b f(x) dx = Q(f) + E(f) \quad (1)$$

$Q(f)$ so-called quadrature formula,
 $E(f)$ error \rightarrow unknown (obvious)

Aim: estimate magnitude of error

so far: error calculated from our knowledge of the exact result

- Obvious: for constant integrand f is $E = 0$, i.e. F_n is independent of n (and always the same)
- Idea: try to estimate the error with help of the *standard deviation* σ :

$$\sigma^2 = \langle f(x)^2 \rangle - \langle f(x) \rangle^2 \quad (2)$$

$$\langle f(x) \rangle = \frac{1}{n} \sum_{i=1}^n f(x_i) \quad (3)$$

$$\langle f(x)^2 \rangle = \frac{1}{n} \sum_{i=1}^n f(x_i)^2 \quad (4)$$

- if f constant $\rightarrow \sigma = 0$

- consider the example $f(x) = 4\sqrt{1-x^2}$ with $F = \int_0^1 f(x)dx = \pi$
Calculate σ for different n (cf. Gould et al. 1996)

F_n	n	$E = F_n - \pi $	σ
3.271771	10^1	0.13017	0.78091
3.100276	10^2	0.04131	0.91441
3.173442	10^3	0.03185	0.85013
3.135863	10^4	0.00572	0.90317
3.142189	10^5	0.00059	0.89051
3.141798	10^6	0.00020	0.89236

- σ almost constant and much larger than E
- but: decrease of E from $n = 10^2$ to $n = 10^4$ by a factor of 10 $\rightarrow \sim 1/n^{1/2}$ (?)
- therefore: σ says how much f varies in $[a, b]$

- idea: estimate E by several *runs* α for constant $n = 10^4$, each with result M_α :

M_α	α	$E = F - \pi $	$ M_{\alpha+1} - M_\alpha $
3.14892	1	0.00735	0.00845
3.13255	2	0.00904	0.01637
3.14042	3	0.00117	0.00787
3.14600	4	0.00441	0.00558
3.15257	5	0.01098	0.00657
3.13972	6	0.00187	0.01285
3.13107	7	0.01052	0.00865
3.13585	8	0.00574	0.00478
3.13442	9	0.00717	0.00143
3.14047	10	0.00112	0.00605

- E varies, differences $|M_\alpha - M_\beta|_{\alpha \neq \beta}$ between results comparable with E , therefore:

Numerical integration and error V

- define standard deviation σ_m of the means:

$$\sigma_m^2 = \langle M^2 \rangle - \langle M \rangle^2 \quad (5)$$

$$\langle M \rangle = \frac{1}{m} \sum_{\alpha=1}^m M_{\alpha} \rightarrow \langle M^2 \rangle = \frac{1}{m} \sum_{\alpha=1}^m M_{\alpha}^2 \quad (6)$$

$$(7)$$

- for the runs 1 till 10 one gets $\sigma_m = 0.006762 \rightarrow$ comparable with E
- exact: one run has the chance of 68% that M_{α} is in in the range $\pi \pm \sigma_m$
- however method not very usefull, as several runs are required
- actually for large n holds:

$$\sigma_m = \frac{\sigma}{\sqrt{n-1}} \approx \frac{\sigma}{\sqrt{n}} \quad (8)$$

e.g., for $n = 10^4$ is $\sigma_m = 0.90317/100 \approx 0.009$, i.e., consistent with our estimate $\sigma_m = 0.007$ and the error $E = 0.006$

How can we get σ without α runs?

Hence, split one run, e.g., in $s = 10$ subsets k such that each contains $n/s = 1000$ trials and has result S_k

Then, with the mean $\langle S \rangle$ from the different runs is also

$$\sigma_s^2 = \langle S^2 \rangle - \langle S \rangle^2 \quad (9)$$

and

$$\sigma_m = \sigma_s / \sqrt{s} \quad (10)$$

Derivation/proof:

- random variable x
- m runs with each n trials ($= m \times n$ trials in total)
- index α labels a run, i a single trial

result from one run (= measurement):

$$M_{\alpha} = \frac{1}{n} \sum_{i=1}^n x_{\alpha,i} \quad (11)$$

the arithmetic mean of all mn trials is:

$$\overline{M} = \frac{1}{m} \sum_{\alpha} M_{\alpha} = \frac{1}{nm} \sum_{\alpha=1}^m \sum_{i=1}^n x_{\alpha,i} \quad (12)$$

difference of a one run α and the total mean

$$e_{\alpha} = M_{\alpha} - \overline{M} \quad (13)$$

Hence the variance (standard deviation²) can be written for the runs as:

$$\sigma_m^2 = \frac{1}{m} \sum_{\alpha=1}^m (M_{\alpha} - \overline{M})^2 = \frac{1}{m} \sum_{\alpha=1}^m e_{\alpha}^2 \quad (14)$$

Now finding the relation between σ_m and σ of the individual $m \times n$ trials. Difference between one trial and the the mean of one *run*:

$$d_{\alpha,i} = x_{\alpha,i} - \overline{M} \quad (15)$$

Therefore the variance for *all* $m \times n$ trials:

$$\sigma^2 = \frac{1}{mn} \sum_{\alpha=1}^m \sum_{i=1}^n d_{\alpha,i}^2 \quad (16)$$

With help of Eq. (15) the Eq. (13) can be rewritten as:

$$e_{\alpha} = M_{\alpha} - \overline{M} = \frac{1}{n} \sum_{i=1}^n (x_{\alpha,i} - \overline{M}) = \frac{1}{n} \sum_{i=1}^n d_{\alpha,i} \quad (17)$$

Insert Eq. (17) into Eq. (14):

$$\sigma_m^2 = \frac{1}{m} \sum_{\alpha=1}^m \left(\frac{1}{n} \sum_{i=1}^n d_{\alpha,i} \right) \left(\frac{1}{n} \sum_{j=1}^n d_{\alpha,j} \right) \quad (18)$$

The products in Eq. (18) consist of terms $i = j$ and terms $i \neq j$. As the trials are independent of each other, for *large* n the differences $d_{\alpha,i}$ and $d_{\alpha,j}$ are on average as often negative as positive, i.e., the terms $i \neq j$ cancel out on average. What remains are the terms for $i = j$:

$$\sigma_m^2 = \frac{1}{mn^2} \sum_{\alpha=1}^m \sum_{i=1}^n d_{\alpha,i}^2 \quad (19)$$

By comparison with Eq. (16) for individual variance: $\sigma^2 = \frac{1}{mn} \sum_{\alpha=1}^m \sum_{i=1}^n d_{\alpha,i}^2$ one gets required variance of runs:

$$\sigma_m^2 = \frac{\sigma^2}{n} \Rightarrow \underline{\underline{\sigma_m = \frac{\sigma}{\sqrt{n}}}} \quad (20)$$

□

→ the standard deviation (= error estimate) scales with $\frac{1}{\sqrt{n}}$

Why Monte-Carlo (integration)?

Performance of integration techniques I

Already seen: for 1d integration, dependence of truncation error on number of intervals (\sim samples)

method	$\sigma(N)$
rectangular rule	N^{-1}
trapezoid rule	N^{-2}
Simpson's rule	N^{-4}
MC sample-mean method	$N^{-1/2}$

→ for 1d MC sample-mean inefficient integration method

Truncation error derived from Taylor series expansion of integrand $f(x)$:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(x_i)(x - x_i)^2 + \dots \quad (21)$$

$$\int_{x_i}^{x_{i+1}} f(x)dx = f(x_i)\Delta x + \frac{1}{2}f'(x_i)(\Delta x)^2 + \frac{1}{6}f''(x_i)(\Delta x)^3 + \dots \quad (22)$$

Performance of integration techniques II

For the **rectangular rule** ($f(x_i)\Delta x$), error Δ_i in leading order for $[x_i, x_{i+1}]$ is

$$\Delta_i = \left[\int_{x_i}^{x_{i+1}} f(x) dx \right] - f(x_i)\Delta x \approx \frac{1}{2}f'(x_i)(\Delta x)^2 \quad (23)$$

→ error per interval; as there are N intervals in total and $\Delta x = (b - a)/N \rightarrow$ total error for rectangular rule $N \Delta_i \sim N (\Delta x)^2 \sim N \left(\frac{b-a}{N}\right)^2 \sim N^{-1}$

Analogously for **trapezoid rule**, where we estimate $f(x_{i+1})$ by Eq. (21):

$$\Delta_i = \left[\int_{x_i}^{x_{i+1}} f(x) dx \right] - \frac{1}{2}[f(x_i) + f(x_{i+1})]\Delta x \quad (24)$$

$$= \left[f(x_i)\Delta x + \frac{1}{2}f'(x_i)(\Delta x)^2 + \frac{1}{6}f''(x_i)(\Delta x)^3 + \dots \right] \quad (25)$$

$$- \frac{1}{2}\Delta x \left[f(x_i) + f(x_{i+1}) + f'(x_i)\Delta x + f'(x_{i+1})\Delta x + \frac{1}{2}f''(x_i)(\Delta x)^2 + \dots \right] \quad (26)$$

$$\approx -\frac{1}{3}f''(x_i)(\Delta x)^3 \rightarrow \text{total error} \sim N^{-2} \quad (27)$$

For **Simpson's rule** $f(x)$ is approximated as parabola on $[x_{i-1}, x_{i+1}] \rightarrow$ terms $\sim f''$ cancel, moreover because of symmetry terms $\sim f'''(\Delta x)^4$ cancel \rightarrow error for interval $[x_i, x_{i+1}]$ is $\sim f^{(4)}(x_i)(\Delta x)^5$ and total error for $[a, b]$ is $\sim N^{-4}$

Integration error in 2d

extend previous estimates for **rectangular rule** in 2d, so for $f(x, y)$: integral \rightarrow sum of volumes of parallelograms with cross section area $\Delta x \Delta y$ and height $f(x, y)$ at one corner

Taylor series expansion of $f(x, y)$

$$f(x, y) = f(x_i, y_i) + \frac{\partial f(x_i, y_i)}{\partial x}(x - x_i) + \frac{\partial f(x_i, y_i)}{\partial y}(y - y_i) + \dots \quad (28)$$

$$\Delta_i = \left[\int \int f(x, y) dx dy \right] - f(x_i, y_i) \Delta x \Delta y \quad (29)$$

Performance of integration techniques IV

Now, substitute Taylor expansion Eq. (28) into error estimate Eq. (29), integrate each term

→ term $\sim f$ cancels out

and $\int (x - x_i) dx = \frac{1}{2}(\Delta x)^2 \rightarrow \int dy$ gives another factor Δy ; similar for $(y - y_i)$

As $O(\Delta y) = O(\Delta x)$, error for interval $[x_i, x_{i+1}]$ and $[y_i, y_{i+1}]$ is

$$\Delta_i \approx \frac{1}{2}[f'_x(x_i, y_i) + f'_y(x_i, y_i)](\Delta x)^3 \quad (30)$$

→ error for one parallelogram $\sim (\Delta x)^3$, for N parallelograms $N \cdot (\Delta x)^3$

But in 2d: $N = A/(\Delta x)^2$

→ total error $N(\Delta x)^3 = N A^{3/2} N^{-3/2} \sim N^{-1/2}$ (whereas in 1d: N^{-1})

Analogously for trapezoid rule in 2d: N^{-1} , for Simpson's rule in 2d: N^{-2}

In general: if in 1d integration error $\sim N^{-p}$

→ integration error in d dimensions $\sim N^{-p/d}$ (curse of dimensionality)

In contrast: MC integration error $\sim N^{-1/2}$ independent of $d \rightarrow$ superior for large d

(think about integrals $\int_V \int_{V_p} f dp^3 dx^3$ in statistical mechanics)

How to integrate in higher dimensions I

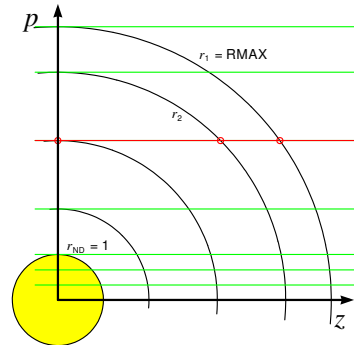
Integrals of functions of more than 1 variable, over regions with $d > 1$, are difficult!

- ① function evaluation: if n function calls required for some accuracy in 1d $\rightarrow \sim n^d$ samples needed for d dimensions (e.g., 30 calls in 1d vs. approx. 30 000 in 3d)
- ② integration region in d dimensions defined by $d - 1$ dimensional boundary \rightarrow can be very complicated for $d > 1$ (e.g. not convex, not simply connected)

Ad 1.) \rightarrow try to reduce integral to lower dimensions by exploiting symmetry of function and boundary and changing coordinates. E.g., spherically symmetric function over spherical region \rightarrow in polar coordinates 1d integral

Example: PoWR code for expanding atmospheres

- non-LTE (i.e. $\vec{n}(\vec{J})$ from statistical equations + ALL \rightarrow Newton's method) radiative transfer in wind (i.e. CMF RT with Mio. of frequency points K , coarsend $\vec{J}(\vec{n})$ for $\vec{n} \rightarrow K \approx 1000$) \rightarrow iteratively solved
- assuming spherical symmetry with, e.g., $ND = 50$ depth-points, typically for each iteration ≈ 5 s, in total ≈ 1000 iterations $\rightarrow \sim h$
- in 3D: $2500 \times$ more “depthpoints” \rightarrow each iteration now 3.5 h (!) \rightarrow total $\frac{1}{2} a$



For each depthpoint: Solve $\mathbf{nP} = 0$ where $P_{ij} := -\sum_{j \neq i}^n P_{ij}$ with $n \sim 500$ + radiative transfer (see sketch)

Ad 2.)

- if boundary complicated, integrand not strongly peaked in very small regions, relatively low accuracy required \rightarrow MC integration! (see below)
- if boundary simple, smooth integrand, (+ high accuracy required) \rightarrow repeated 1d integrals or multidimensional quadrature
- if integrand peaks in certain regions \rightarrow split integral into several “smooth” regions (requires knowledge of behaviour of integrand)

How to integrate in higher dimensions IV

Repeated 1d integration

Let $d = 3$ with x, y, z and boundaries $[x_1, x_2]$, $[y_1(x), y_2(x)]$, $[z_1(x, y), z_2(x, y)] \rightarrow$ find x_1, x_2 and functions $y_1(x), y_2(x), z_1(x, y), z_2(x, y)$ such that

$$\int \int \int dx dy dz f(x, y, z) = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x, y)}^{z_2(x, y)} dz f(x, y, z) \quad (31)$$

Example: 2d integral over circle with radius R centered on $(0, 0)$

$$\int_{x_1=-R}^{x_2=+R} dx \int_{y_1(x)=-\sqrt{R-x^2}}^{y_2(x)=\sqrt{R-x^2}} dy f(x, y) \quad (32)$$

Note that Fubini's theorem for iterated integrals assumes that the integrand is absolutely integrable:

$$\int \int |f(x, y)| dx dy < +\infty.$$

How to integrate in higher dimensions V

Innermost integration over z yields a function $G(x, y)$:

$$G(x, y) := \int_{z_1(x, y)}^{z_2(x, y)} f(x, y, z) dz \quad (33)$$

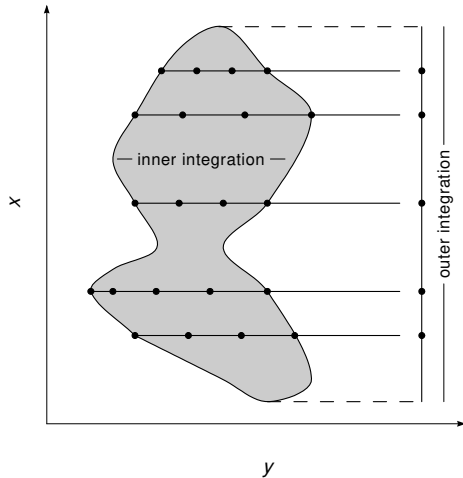
then integration over y yields $H(x)$:

$$H(x) := \int_{y_1(x)}^{y_2(x)} G(x, y) dy \quad (34)$$

finally the overall integral I is

$$I = \int_{x_1}^{x_2} H(x) dx \quad (35)$$

How to integrate in higher dimensions VI



instead of using fixed Cartesian mesh of points, better evaluate function at suitable x locations (along y -axis), while inner integration (over y) chooses suitable y values;
→ inner integration call (over y) many more times than outer integration (over x)

Implementation of Eq. (33)-(35) requires 3 separate copies of some 1d integration routine, so one for each x, y, z integration or recursive calls of the same routine

Example: Fortran snippet for 3d iterated integration

```
! identical copies quadx, quady, quadz  
! of 1d-integration routine;  
! user provides func(x,y,z), y1(x),  
! y2(x), z1(x,y), z2(x,y) as in Eq. (31)
```

```
SUBROUTINE quad3d(x1, x2, ss)  
  REAL ss, x1, x2, h  
  CALL quadx(h, x1, x2, ss)  
  RETURN  
END
```

```
FUNCTION f(zz)  
  REAL f, zz, func, x, y, z  
  COMMON /xyz/ x, y, z  
  z = zz  
  f = func(x, y, z)  
  RETURN  
END
```

```
FUNCTION g(yy)  
  REAL g, yy, f, z1, z2, x, y, z  
  COMMON /xyz/ x, y, z  
  REAL ss  
  y = yy  
  CALL quadz(f, z1(x,y), z2(x,y), ss)  
  g = ss  
  RETURN  
END
```

```
FUNCTION h(xx)  
  REAL h, xx, g, y1, y2, x, y, z  
  COMMON /xyz/ x, y, z  
  REAL ss  
  x = xx  
  CALL quady(g, y1(x), y2(x), ss)  
  h = ss  
  RETURN  
END
```


Example: Mass and center of mass of cut torus

Section of a torus with radius R and cross section radius r

$$z^2 + (\sqrt{x^2 + y^2} - R)^2 \leq r^2 \quad (36)$$

section defined by

$$x \geq a \quad y \geq b \quad (37)$$

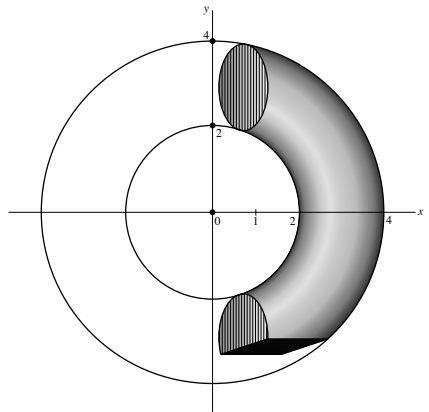
Need to evaluate following integrals

$$M = \int \rho \, dx \, dy \, dz \quad M_x = \int x \rho \, dx \, dy \, dz \quad (38)$$

$$M_y = \int y \rho \, dx \, dy \, dz \quad M_z = \int z \rho \, dx \, dy \, dz \quad (39)$$

i.e., x-coordinate of center of mass is $x = M_x/M$ and so on

MC integration for a torus (centered on origin, outer radius = 4, inner radius = 2) section, where $x \leq 1$ and $y \leq -3$, i.e., bounds given by intersection of two planes. Integration limits cannot be easily given in analytically closed form



from Press et al. (2007)

Choose region that encloses torus section, e.g, rectangular box with $1 \leq x \leq 4$, $-3 \leq y \leq 4$, and $-1 \leq z \leq 1$, hence total volume of box is $V = 3 * 7 * 2$

Example: C/C++ snippet for MC integration of torus section

```
int N = 1000 ; // sample points
double V = 3. * 7. * 2. ; // sample volume
double den = 1. ; // density rho
double sw = 0., varw = 0. ; // mass and variance
double swx = 0., varx = 0. ; // x-coordinate and var. for center of mass
...
for (i = 0 ; i < N ; ++i) {
x = 1. + 3. * rand()/double(RAND_MAX) ; // cut of torus
y = -3. + 7. * rand()/double(RAND_MAX) ; // cut of torus
z = -1. + 2. * rand()/double(RAND_MAX) ;
if ( pow(z*z + (sqrt(x*x + y*y) -3. ), 2.) <= 1. ) {
    sw = sw + den ; varw = varw + den*den ;
    swx = swx + x * den ; varx = varx + (x*den)*(x*den) ;
    ...
} }
w = V * sw / N ; // mass of torus
x = V * swx / N ; // x-coordinate
dw = V * sqrt((varw / N - (sw/N)*(sw/N)) / N) ; // error estimate mass
dx = V * sqrt((varx / N - (swx/N)*(swx/N)) / N) ; // error estimate x-coordinate
...
```

Conclusions about advantage of MC integration

- ① MC integration error decreases independent of dimension with $\sim N^{-1/2} \rightarrow$ superior for integrals with many integration variables (e.g., phase space integrals, QM)
- ② MC integration easy to implement for any geometry \rightarrow superior for 3d models without simple symmetry (e.g., spherical symmetry)

Techniques of MC parallelization

Neutron transport with packets I

So far: single neutron n^0

Improvement/speed up: consider “neutron packets”, i.e. we follow an ensemble of neutrons (which advances with random $\ell, \cos \theta$ as before)

→ determine *fraction* of the scattered and captured neutrons

1. scattering: fraction of scattered n^0 : p_s , fraction of absorbed n^0 : p_c
2. scattering: fraction of scattered n^0 : p_s^2 , fraction of absorbed n^0 : $p_c p_s$
- m th scattering: fraction of scattered n^0 : p_s^m , fraction of absorbed n^0 : $p_c p_s^{m-1}$

so, after m th scattering:

→ total fraction of captured neutrons:

$$f_c = p_c + p_c p_s + p_c p_s^2 + \dots + p_c p_s^{m-1}$$

→ total fraction of scattered neutrons:

$$f_s = p_s^m$$

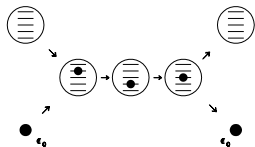
→ if position $x < 0$: add f_s to f_{refl}

→ if position $x > t$: add f_s to f_{trans}

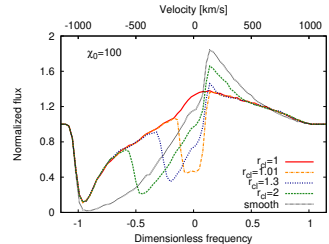
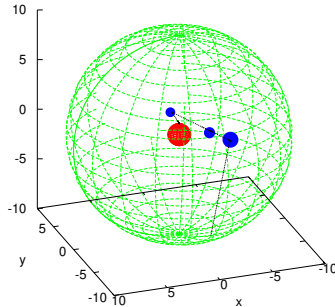
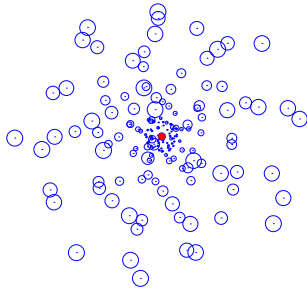
→ Note: requires normalization of the fractions afterwards

→ see: Lucy (2002): “Monte Carlo transition probabilities”

- instead of individual photons, use energy packets of photons of same frequency ν ($\epsilon(\nu) = nh\nu$), packets always have same energy $\epsilon_0 \rightarrow$ different n
- elastic scattering (e.g., Thomson, resonance): $\nu_e = \nu_a$
- absorption leads to re-emission following: $\epsilon(\nu_e) = \epsilon(\nu_a)$, no packet (= energy) lost or created \rightarrow divergence-free radiation field
- macro-atoms with discrete internal states, activation via r-packet (radiative) of appropriate CMF frequency or k-packet (kinetic); active macro-atom performs internal transitions and gets inactive by emission of r- or k-packet



→ see Šurlan et al. (2012): “Three-dimensional radiative transfer in clumped hot star winds. I. Influence of clumping on the resonance line formation”



2-D projection of an example of a realization of a stochastic 3-D wind model. The path of one particular photon inside a realization of our clumped wind. The effect of variation of the onset of the clumping r_{cl} on a resonance line

Parallelization

Many runs in MC simulations required for reliable conclusions ($\sigma \sim \frac{1}{\sqrt{N}}$)

Often: Result of one run (e.g., path of a neutron through a plate) *independent* from other runs

→ **Idea:** acceleration by parallelization

Problem: concurrent access to memory resources, i.e. variables (e.g., n_s , f_{refl})

Solution: special libraries that enable multithreading (e.g., OpenMP) or multiple processes (e.g., MPI) for one program

→ insert: pipelining, vectorization, parallelization

What influences the performance of a CPU (= runtime of your code)?

- architecture/design: out-of-order execution (all x86 except for Intel Atom), pipelining (stages), vectorization units (width)
- **cache sizes** (kB ... MB) and location: L1 cache for each core, L3 for processor
- clock rate (\sim GHz): only within a processor family usable for comparison due to different number of instruction per clock (IPC) of design, even more complicated because of variable clock rates (base, peak) to exploit TDP (thermal design power)
→ impact on single-thread performance
- number of cores (1 ...): → impact on multi-thread performance

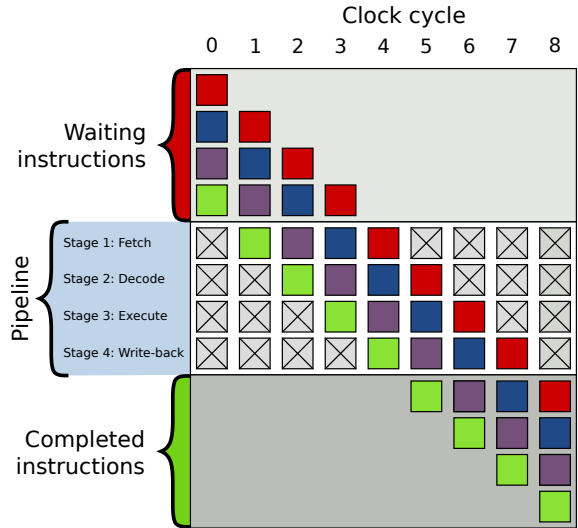
Pipelining I

splitting machine instruction into a sequence

independent execution of instructions, each consisting of

- instruction fetching (IF)
- instruction decoding (ID) + register fetch
- execution (EX)
- write back (WB)

operations of instructions are processed at the same time → quasi parallel execution, higher throughput



By en:User:Cburnett - Own workThis vector image was created with Inkscape., CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=1499754>

NetBurst disaster

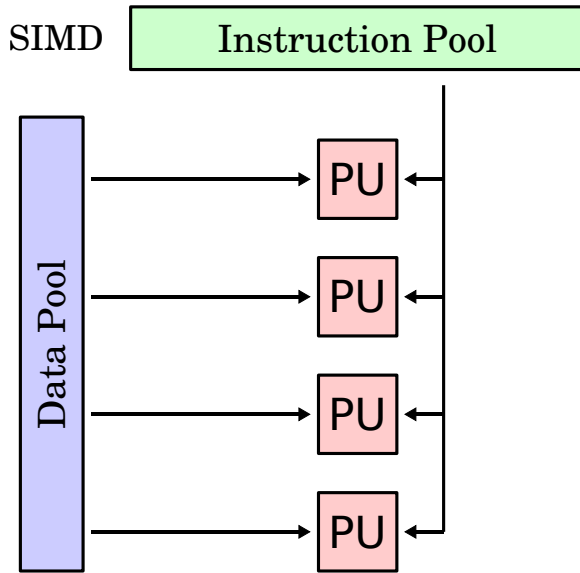
Pentium 4 (2000-2008) developed to achieve $> 4\text{GHz}$ (goal: 10 GHz) clockrate by several techniques, i.a., *long* pipeline:

- 20 stages (Pentium III: 10) up to 31 stages (Prescott core)
- smaller number of instructions per clock (IPC) (!)
- increased branch misprediction (also only 10%, improved by 33% for Pentium III)
- larger penalty for misprediction

→ compensated by higher clock rate

higher clock rate → higher power dissipation, especially for 65 (Presler, Pentium D), 90 (Prescott) up to 180 nm (Williamette) structures

→ power barrier at 3.8 GHz (Prescott)



- **SSE** - Streaming SIMD Extensions (formerly: ISSE - Internet SSE)
- **SIMD** - Single Instruction Multiple Data (→ cf. Multivec, AMD3Dnow!), introduced with Pentium III (Katamai, Feb. 1999)

- enables vectorization of instructions (not to be confused with pipelining or parallelization), often new, complex machine instructions required, e.g., PANDN \rightarrow bitwise NOT + AND on *packed integers*
- comprises 70 different instructions, e.g., ADDPS – add packed single-precision floats (two “vectors” each with 4 32 bit) into a 128 bit register
- works with 128 bit registers (3Dnow! only 64 bit), but first execution units (before Core architecture) only with 64 bit
- **AVX** - Advanced Vector Extensions with 256 bit registers, theoretically doubled speed! since Sandy Bridge (Intel Core 2nd generation, e.g., i7-2600 K) and Bulldozer (AMD) \rightarrow AVX-512 with 512 bit registers in Skylake (6th generation, e.g., Core i7-6700); AMD Zen 4
Note: AVX-512 instructions may reduce the clockrate on Intel CPUs (heat limit)

- supported by all common compilers, e.g.,
ifort -sse4.2
ifort -axcode COMMON-AVX512
g++ -msse4.1
g++ -mavx512f
- very easy (automatic) and efficient optimization, e.g., for unrolled loops → vectorization

Caution!

Different precisions for SSE-doubles (e.g., 64 bit) and FPU-doubles (80 bit), especially for buffering, so results of doubles, e.g.,

```
xx = pow(x,2) ;
```

```
sqrt( xx - x*x) ;
```

usually not predictable

Mult-cores

- originally one core per processor, sometimes several processors per machine/board (supercomputer)
- many units, e.g., arithmetic logic unit (ALU), register, already multiply existing in one processor
- first multi-core processors: IBM POWER4 (2001); desktop → Smithfield (2005), e.g., Pentium D
- Hyper-threading (HT): introduced in Intel Pentium 4 → for better workload of the computing units by simulation of another, logical processor core (compare: AMD Bulldozer design with modules)
- today: up to 64 cores for desktop (AMD Zen: Ryzen Threadripper 5995WX, TDP 280 W) or 96 for servers (e.g., AMD EPYC 9654, TDP 360 W – even 2 CPUs per board) + Hyperthreading
- arms race of cores instead of clock rate (NetBurst disaster)

Acceleration by parallelization

- parallelization done, e.g., by multithreading (from *thread*)
for shared memory (RAM on one “node”, usually on one mainboard)
- “The free lunch is over” → no simple acceleration more of *single-thread* programs by pure increase of clock rate (exceptions: Turbo Boost, Turbo Core, in some ways larger caches may help)
- multithreading supported by, e.g., **OpenMP** (shared memory), see below
- different from: multiprocessing parallelization via **MPI** (Message Passing Interface)
→ distributed computing (cf. Co-array Fortran) but can be combined: MPI + OpenMP;
usually: MPI more complicated (and slower) than OpenMP → trend for “larger nodes”

General-purpose computing on graphics processing units → further development of graphic cards

- e.g., Nvidia (Tesla, Fermi); AMD (Radeon Instinct)
→ *Frontier* (USA, 1st since June 2022 in Top500) with 9 472 nodes (each with AMD-EPYC-7A53 64core CPU + 4 GPU MI250X x2) reaches 1.1 ExaFLOPS (for comparison: 24 core desktop CPU \approx 8 TeraFLOPS $\rightarrow 7 \times 10^{-6}$ of *Frontier*)
- so-called shaders → highly specialized ALUs, often only with single precision (opposite concept: Intel's Larrabee)
- programming (not only graphics) via CUDA (Nvidia) or OpenCL (more general)
- OpenCL → parallel programming for arbitrary systems, also NUMA (non-uniform memory access), but very abstract and complex concept and also complicated C-syntax
- CUDA support, e.g., by PGI Fortran compiler → simple acceleration without code modifications

OpenMP

OpenMP - Open Multi-Processing

- for shared-memory systems (e.g., multi core) per node
- directly available in g++, gfortran, and Intel compilers
- insertion of so-called OpenMP (pragma) directives :

Example: for loop

C++

```
#include <omp.h>
...
#pragma omp parallel for
for (int i = 1 ; i <= n ; ++i)
{ ... }
```

Fortran

```
USE omp_lib ! ifort declarations
!$OMP PARALLEL DO
DO i = 1, n
....
ENDDO
!$OMP END PARALLEL DO
```

instructs parallel execution of the for loop, i.e., there are copies of the loop (different iterations) which run in parallel

→ only the labeled section runs in parallel

→ pragma directives are syntactically seen **comments**, i.e., invisible for compilers without OpenMP support

- realization during runtime by *threads*
- number of used threads can be set, e.g., by environment variable

```
export OMP_NUM_THREADS=4  # bash  
setenv OMP_NUM_THREADS 4  # tcsh
```

→ obvious: per core only one thread can run at the same time (but: Intel's hyper-threading, AMD's Bulldozer design) → in HPC often reasonable:

number of threads = number of physical CPU cores

Caution!

Distributing and joining of threads produces some overhead in CPU / computing time (e.g., copying data) and is therefore only efficient for complex tasks within each thread. Otherwise multithreading can slow down program execution.

Including the OpenMP library:

C++

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

Fortran

```
!      only needed for declaration of
!      OMP functions etc. with ifort:
!$     use omp_lib
```

→ instructions between `#ifdef _OPENMP` and `#endif` (Fortran: following `!$`) are only executed if compiler invokes OpenMP

Compile with

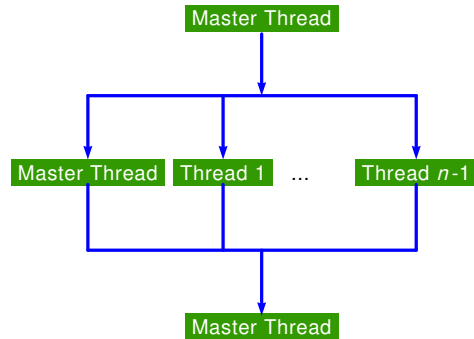
<code>g++</code>	<code>-fopenmp</code>	
<code>icpx</code>	<code>-qopenmp</code>	(deprecated: <code>-openmp</code>)
<code>gfortran</code>	<code>-fopenmp</code>	
<code>ifort</code>	<code>-qopenmp</code>	(deprecated: <code>-openmp</code>)

OMP functions

Useful: functions specific for OpenMP, e.g., for number of available CPU cores, generated (maximum) number of threads, and current number of threads:

```
omp_get_num_procs()    // number of (logical) processor cores
omp_get_max_threads()  // max. number of (automatic) generated threads
omp_get_num_threads()  // number of current threads
omp_get_thread_num()   // number of the current thread
```

Join-fork model:
thread that executes
parallel directive
becomes master of
thread group with ID= 0



OMP – Access to variables: shared and private I

Very important: organization of the accessibility of the involved data, i.e. assign attributes shared or private to thread variables

shared

→ **default** for variables declared outside the parallel section
data are visible in all threads and can be modified (concurrent access)

```
int sum = 0 ;  
#omp pragma parallel for  
for (int k = kmax ; k > 0 ; --k) {  
    sum += k ; // sum is implicitly shared  
  
    NSUM  = 0  
!$OMP PARALLEL DO  
    DO K = KMAX, 1, -1  
        NSUM = NSUM + K    ! NSUM is implicitly shared
```

in contrast to:

private

each thread has its own copy of the data, which are invisible for other threads, especially from outside of the parallel section.

Loop iteration variables are private by default and should be declared in the loop header for clarity:

```
#omp pragma parallel for  
for (int k = kmax ; k > 0 ; --k)  // k is implicitly private
```

```
!$OMP PARALLEL DO  
    DO K = KMAX, 1, -1                ! K is implicitly private
```

Moreover, there are further so-called data clauses, e.g., `firstprivate` (initialization before the parallel section), `lastprivate` (last completed thread determines the value of the variable after the parallel section) and many more ...

→ This is the complicated part of OpenMP!

OMP – Access to variables: shared and private III

Example private

C++:

```
int j, m = 4 ;  
#pragma omp parallel for private (j)  
for (int i = 0 ; i < max ; i++) {  
    j = i + m ;  
    ... ;  
}
```

Fortran:

```
      INTEGER :: j, m  
!$OMP PARALLEL DO PRIVATE (j)  
      DO i = 0, max  
          j = i + m  
          ...  
      ENDDO  
!$OMP END PARALLEL DO
```

→ loop variable *i* and explicitly private variable *j* as “local” copies in each thread

→ variable *m* implicitly shared (be careful in Fortran because of implicit declarations within, e.g. loops)

General form of OpenMP directive for parallelization:

```
#pragma omp parallel
```

→ parallel section also possible without a loop, section is executed per thread
(in C/C++: { } block required for multiple commands):

C++:

```
#pragma omp parallel
{
    cout << "Hi!" ;
    cout << endl ;
}
```

Fortran:

```
!$OMP PARALLEL
      print *, "Hi!"
!$OMP END PARALLEL
```

```
#pragma omp critical
```

→ within a parallel section

is executed by each thread, but never at the same time (avoiding race conditions for shared resources)

C++:

```
#pragma omp critical
{
    WDrawPoint(myworld, x, y, c) ;
}
```

Fortran:

```
!$OMP CRITICAL
      CALL PGDRAW (x, y)
!$OMP END CRITICAL
```

Example: critical access to an array

C++:

```
#pragma omp parallel for private (j)
for (int i = 0 ; i < nymax ; ++i) {
    for (j = 0 ; j < nxmax ; ++j ) {
        ...
        #pragma omp critical
        subset[i][j] = result ;
    }
}
```

Fortran:

```
!$OMP PARALLEL DO private (j)
    DO i = 0, nymax - 1
        DO j = 0, nxmax - 1
            ...
            !$OMP CRITICAL
                subset(i,j) = result
            ENDDO
        ENDDO
    ENDDO
```

→ critical forces threads to queue, hence slows down execution, better: if possible, use reduction clause:

OMP – critical and reduction IV

```
#pragma omp parallel reduction (operator:list of variables)
```

The reduction clause defines corresponding (scalar) variables in a parallel section.

Example: summing up with reduction

C++:

```
#pragma omp parallel for \  
  private(x) reduction(+:sum_this)  
for (int i = 1; i <= nmax ; i++) {  
  x = 0.01 / (i + 0.5) ;  
  sum_this += x ;  
}
```

Fortran:

```
!$OMP PARALLEL DO PRIVATE(x)  
!$  > REDUCTION(+:sum_this)  
  DO i = 1, nmax  
    x = 0.01 / (i + 0.5)  
    sum_this = sum_this + x  
  ENDDO
```

There are a number of allowed operators for reduction, e.g.:

operator	meaning	data type	neutral element / initial value
+, -	sum	int, float	0
*	product	int, float	1
&	bitwise and	int	all bits 1

- Heads up! OpenMP needs clear syntax for loop parallelization:

```
for (int i = 0 ; i < n ; i++)
```

make sure that your loop has *canonical loop form*, especially the loop iteration variable (here: i) is integer as well as variables used for comparison (here: n). OpenMP is very picky and might otherwise (e.g., if n is float) stop compilation:

error: invalid controlling predicate.

- Note that `omp parallel for` / `OMP PARALLEL DO` is the contracted form of

C++:

```
#pragma omp parallel
{
    #pragma omp for
    for ( ... ) {
        ...
    }
}
```

Fortran:

```
!$OMP PARALLEL
!$OMP DO
    ...
!$OMP END DO
!$OMP END PARALLEL
```


`schedule(runtime)`

Examples:

```
#pragma omp parallel for schedule (runtime)
```

→ way of distributing the parallel section to threads is defined at runtime, e.g., by (bash)

```
export OMP_SCHEDULE "dynamic,1"
```

→ each thread gets a *chunk* of size 1 (e.g., one iteration) as soon as it is ready

```
export OMP_SCHEDULE "static"
```

→ the parallel section (e.g., loop iterations) is divided by the number of threads (e.g., 4)
and each thread gets a chunk of the same size

→ **static is the default**

Useful for performance measurement:

```
omp_get_wtime() // → returns the so-called wall clock time (not the cpu time)
```

```
omp_get_thread_num() // → returns the number of the current thread
```

Weblinks:

<http://www.openmp.org/>

especially the documentation of the specifications:

<http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf>

Gould, H., Tobochnik, J., & Wolfgang, C. 1996, An Introduction to Computer Simulation Methods: Applications to Physical Systems (2nd Edition) (USA: Addison-Wesley Longman Publishing Co., Inc.)

Lucy, L. B. 2002, A&A, 384, 725

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 2007, Numerical Recipes 3rd Edition: The Art of Scientific Computing, 3rd edn. (Cambridge University Press)

Šurlan, B., Hamann, W. R., Kubát, J., Oskinova, L. M., & Feldmeier, A. 2012, A&A, 541, A37