Introduction to High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025

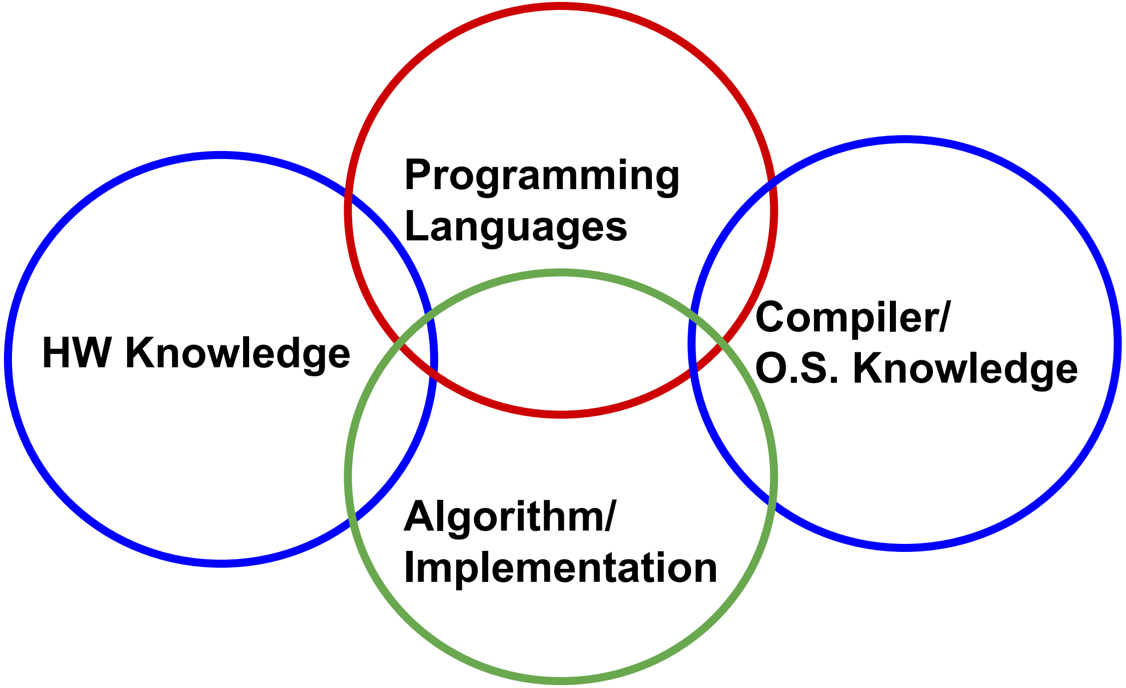
g.amati@cineca.it/g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ **HPC: What is it?**
- ✓ **Hardware: how it works**
- ✓ **Algorithm vs. Implementation**
- ✓ Compiler
- ✓ Parallel Paradigm
- ✓ Conclusions & Comments

HPC: what it is?

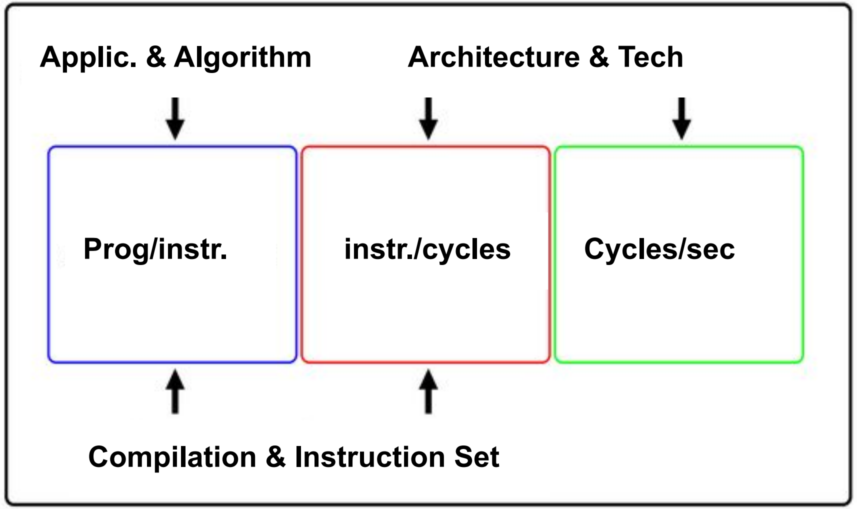
- ✓ These are the main skills for an efficient HPC



Who is in charge of what?

High performance is the joint effort of different “players”

- ✓ **Programmer**: in charge of the choice of the algorithm
- ✓ **Compiler**: in charge of the “language” traduction in instructions
- ✓ **HW**: in charge of executing the instructions



Algorithm

Definition:

a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer…..

- ✓ From previous lessons we have learned that, for a computer, different instructions can have different impact in execution time
- ✓ We are not talking about "correctness" (when time is not an issue) but about "computational complexity"
- ✓ As example
  - O(N^2) << O(N^4) for N big enough (still to define)
- ✓ The choice, at the beginning, of an inefficient algorithm can be really "dangerous"!

Complexity

In computer science, the computational complexity or simply complexity of an algorithm is the amount of resources required to run it. Particular focus is given to computation time (generally measured by the number of needed elementary operations) and memory storage requirements.

- ✓ Each algorithm has its own complexity, i.e. the number of instruction to be performed to complete the work
- ✓ If ***N*** is the size of the problem ***O(N^?)*** is the problem complexity
  - MMM → O(N^3)
  - MMM (Stressen) → O(N^2.80)
- ✓ Hint:
  - Try to understand complexity of your algorithm
  - Verify if your assumption is correct: does it scale as it is supposed to?
  - Stress the algorithm for “real-size” problem: usually development is done using very little problem (in size). to little to be “killed” by high complexity (e.g. cache effects)
  - For little problems size of the prefactor could “hide” real performance

HW Evolution

Hardware evolution (from 1947 to 1985)

✓ From: More Programming Pearls: Confessions of a Coder, Bentley

HW	Year	Mflops
Manchester Mark 1	1947	0.0002
IBM 701	1954	0.003
IBM Stretch	1960	0.3
CDC 6600	1964	2
CDC 7600	1969	5
Cray-1	1976	50
Cray-2	1985	125

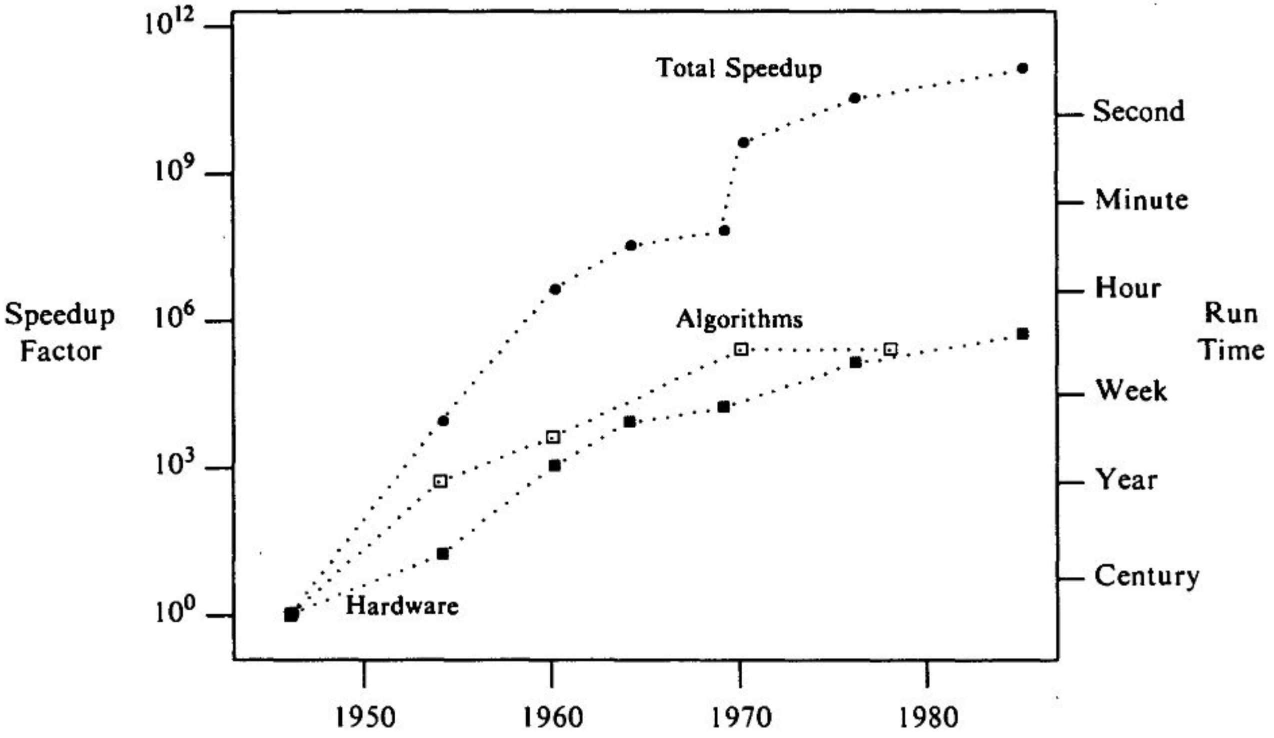
Algorithm Evolution

Algorithm evolution (from 1945 to 1978) for 3D elliptic equation (e.g. pressure)

✓ From: More Programming Pearls: Confessions of a Coder, Bentley

Algorithm	Year	Complexity
Gaussian Elimination	**1947**	**N^7**
SOR (suboptimal)	**1954**	**8N^5**
SOR (optimal)	**1960**	**8N^4*log(N)**
Cyclic Reduction	**1964**	**8N^3*log(N)**
Multigrid	**1969**	**60N^3**

Which is more important?



Example: solution of a linear system

$$Lx = b$$

where

- ✓ L is a nxn lower triangular matrix
- ✓ x is a n-vector
- ✓ b is a n-vector lower triangular matrix of known numbers

Two possible way of solving the system

- ✓ forward substitution
- ✓ matrix partition (?)

Example: forward substitution

$$Lx = b$$

- ✓ Fortran implementation

```
do i = 1, n
    do j = 1, i-1
        b(i) = b(i) - L(i,j)*b(j)
    enddo
    b(i) = b(i)/L(i,i)
enddo
```

- ✓ time (old stuff): 8.06''

Example: matrix partition

$$Lx = b$$

✓ Fortran implementation

```
do j = 1, n
    b(j) = b(j)/L(j,j)
    do i = j+1, n
       b(i) = b(i) - L(i,j)*b(j)
    enddo
enddo
```

✓ time (old stuff): 2.56''

Which the limit?

In HPC it is mandatory to know which is the limit do the optimization process?

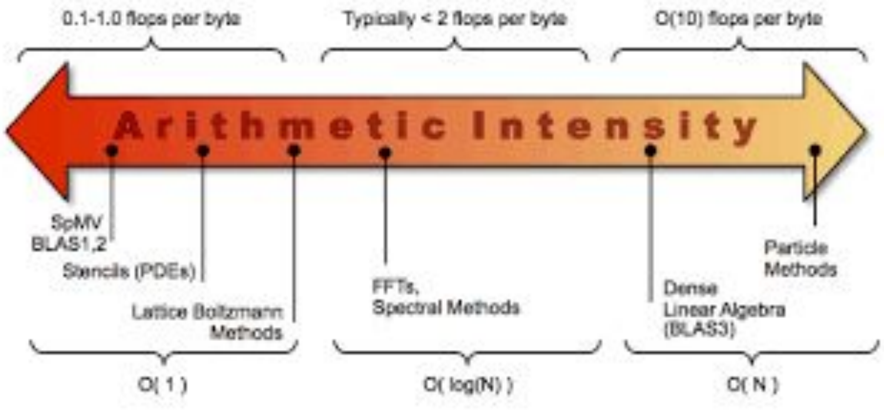
- ✓ How far am I for the limit?
- ✓ Am I within my time constraint?

A HW system has 2 key figures

- ✓ How many floating point operations can I perform? → FLOPS
- ✓ How many data can I move back and forth → BW

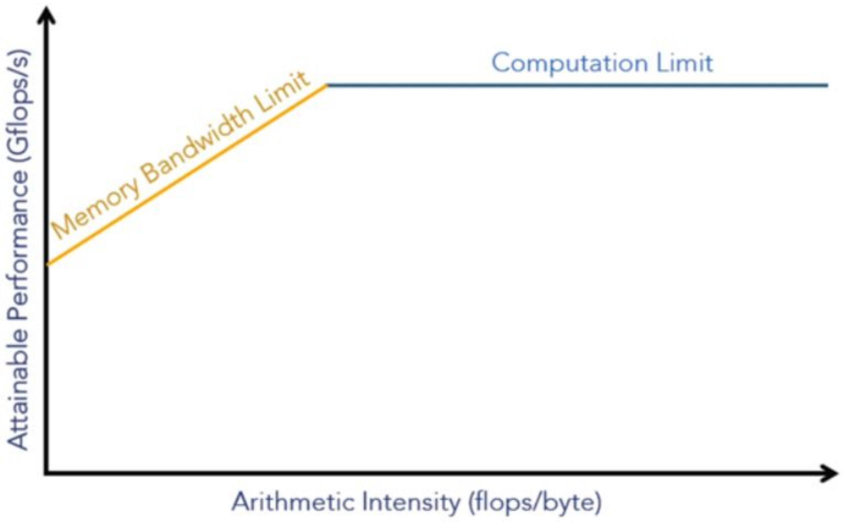
Arithmetic Intensity of a code

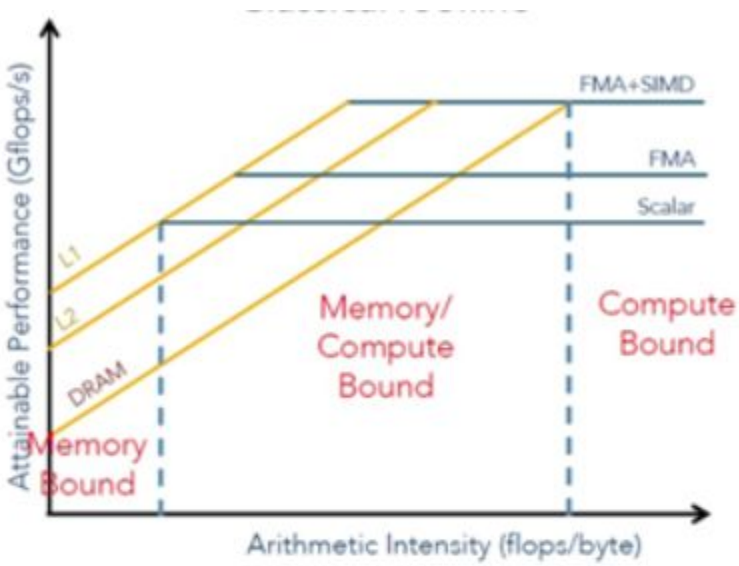
- ✓ A.I = ratio #Flops/#Byte moved



Roofline Model

✓ https://www.nersc.gov/assets/Uploads/Tutorial-ISC2019-Intro-v2.pdf



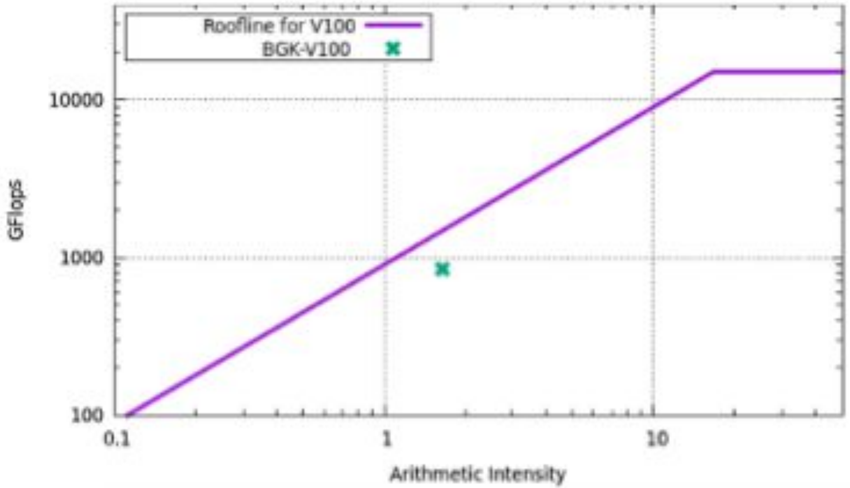


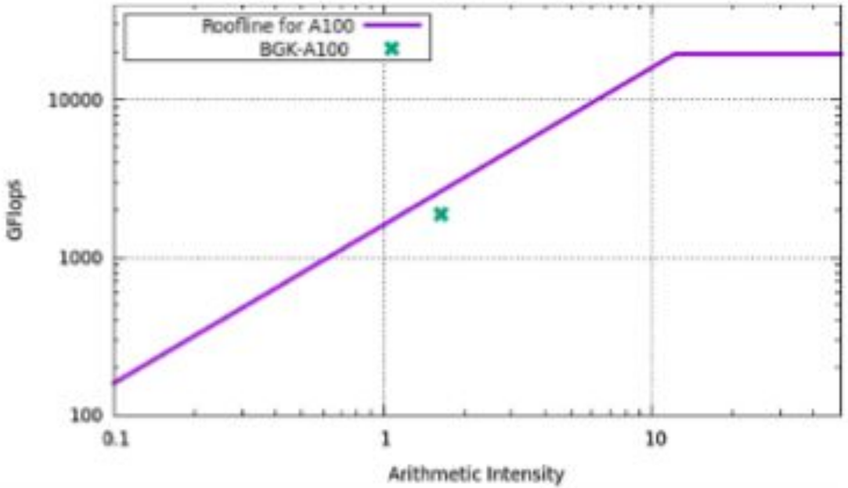
Roofline Model

How to increase performance?

✓ Reduce data movement

✓ Increase Floating point operations (?)





Performance evolution/Intel

CPU (codename)	Clock Frequency	Number of core	Flops cycle (DP)	Peak Perf. (Gflops)
Xeon E5645 (Westmere)	2.4 GHz	2x6	4	115
Xeon E5-2687W0 (S.Bridge)	3.1 GHz	2x8	8	396
Xeon E5-2670v2 (I. Bridge)	2.5 GHz	2x10	8	400
Xeon E5-2630v3 (Hashwell)	2.4 GHz	2x8	16 (AVX-256bit)	614
Xeon E5-2697v4 (Broadwell)	2.3 GHz	2x18	16 (AVX-256bit)	1325
Xeon Platinum (Skylake)	2.1 GHz	2x24	32 (AVX-512bit)	3225

No increase

Factor 4x

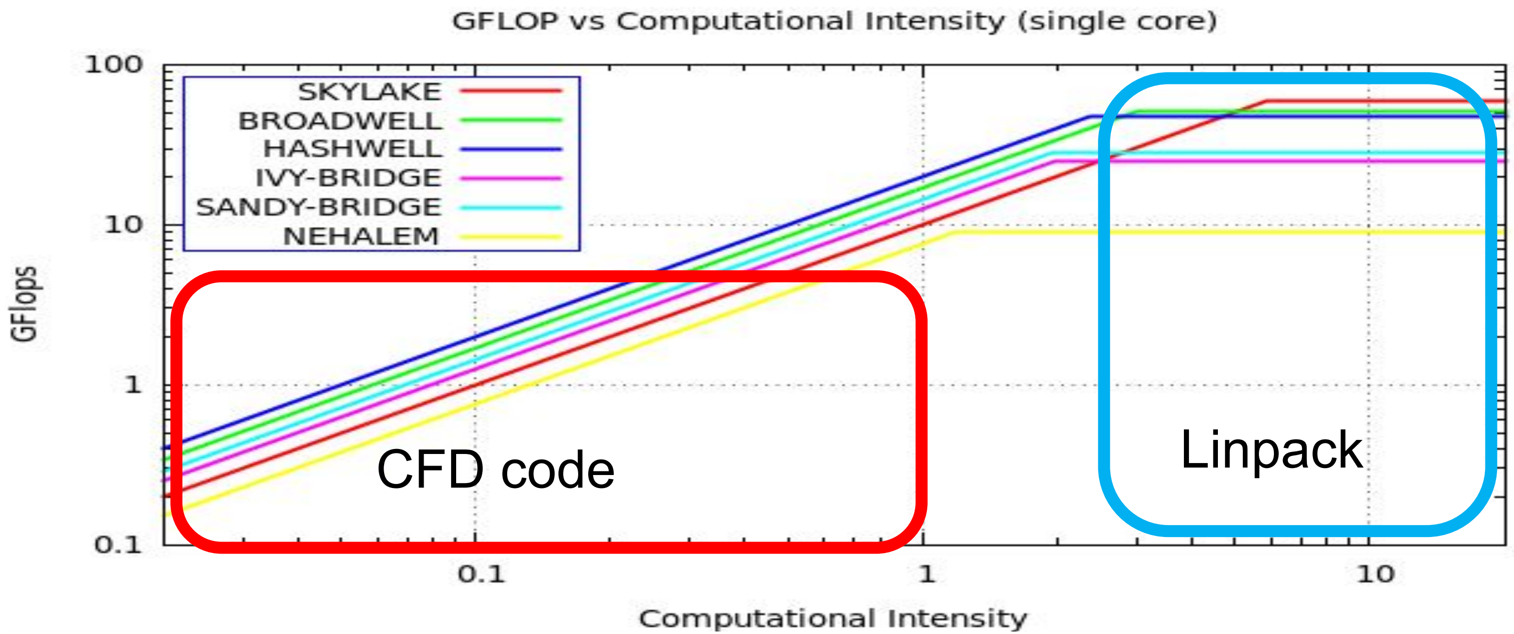
Factor 8x

Total:
Factor 32x

Real performance: serial

Performance ordered with respect to computational intensity (AI) =#flops/#byte

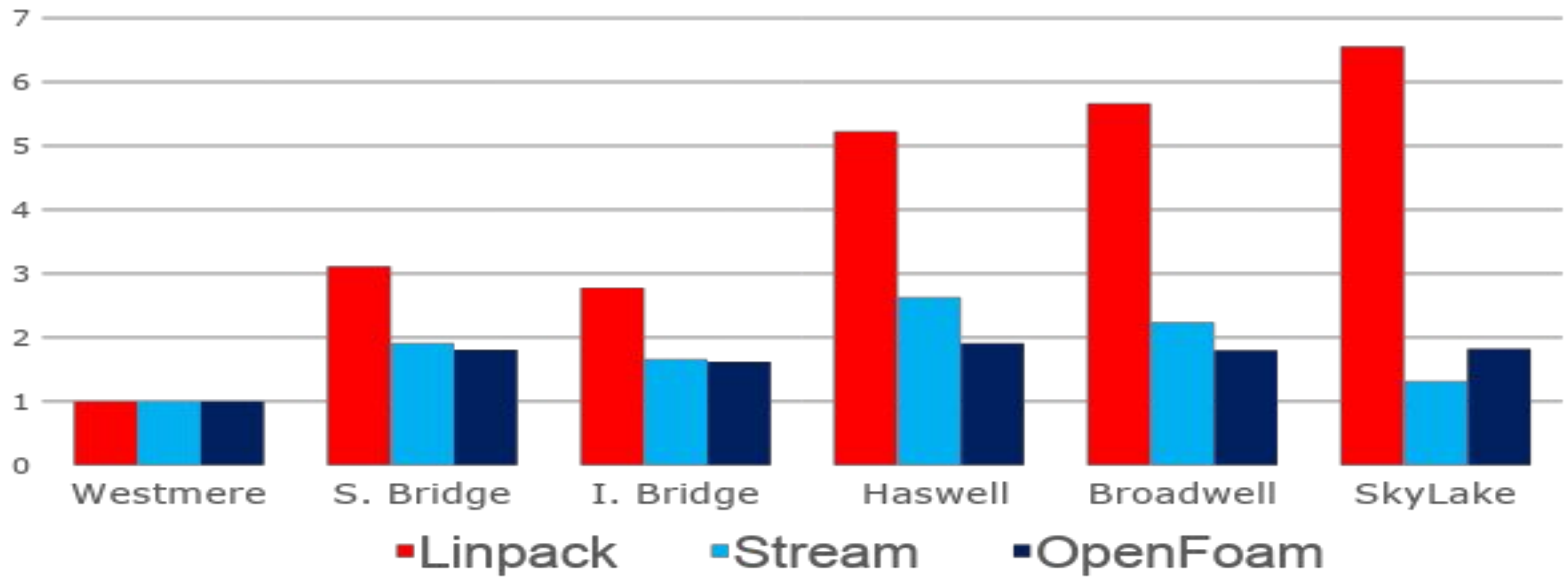
- CI > 1 FLOPs limited
- CI < 1 BW limited



Real performance Improvements

Performance for single core performance

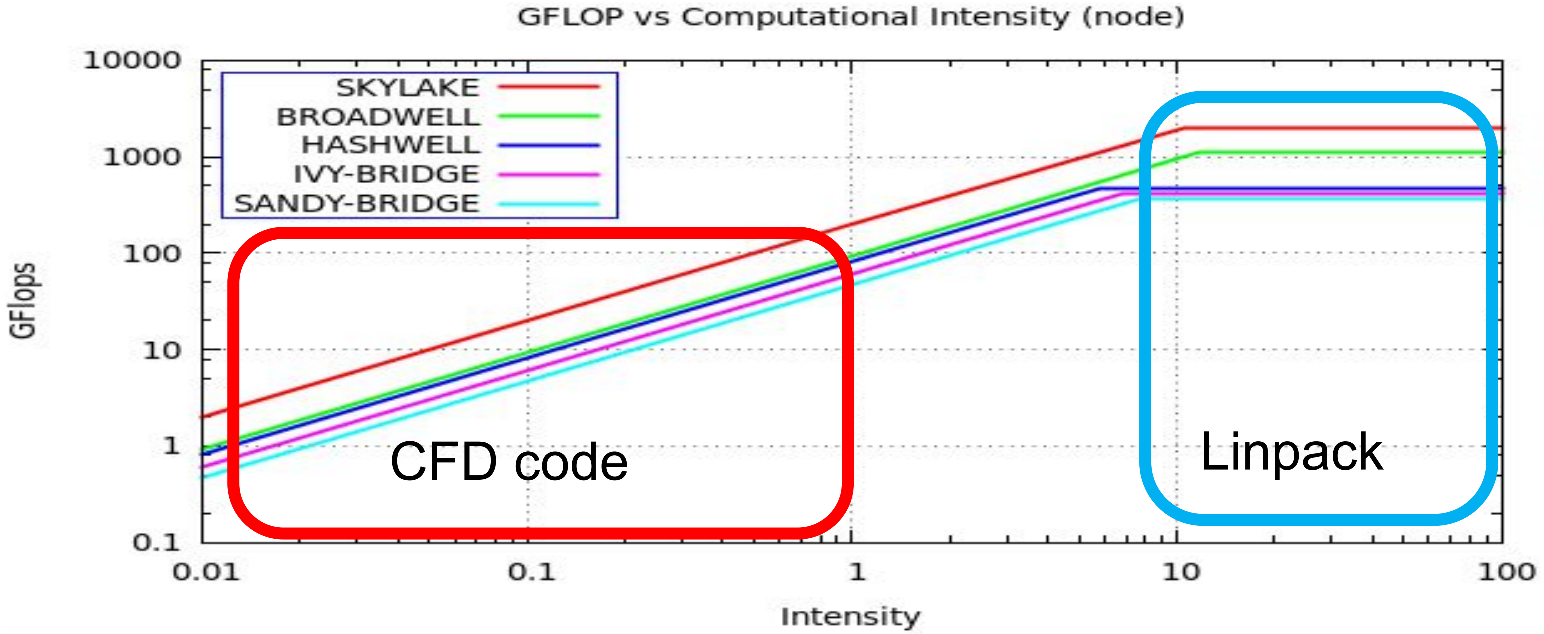
- OpenFoam, 3D lid-driven cavity, 80^3 grid-point, serial
- Linpack
- Stream



Real performance: parallel

Performance ordered with respect to computational intensity (AI): #flops/#byte

- CI > 1 FLOPs limited
- CI < 1 BW limited



Implementation

Implementation is the execution or practice of a plan, a method or any design, idea, model, specification, standard or policy for doing something.

- ✓ Starting from a well defined Algorithm the programmer writes the program
  - Programming languages
  - data allocation
    - data movement
    - I/O
  - compiler, tools
  - HW
- ✓ Each of these points can improve/depress performance: implementation issues

Instrument your code

Hint

- ✓ Instrument your code with timing functions
- ✓ Not all function but “logical” block (e.g. I/O, FFT, init, diagnostic)

```
call SYSTEM_CLOCK(countD0, count_rate, count_max)
// do something //
call SYSTEM_CLOCK(countD1, count_rate, count_max)
time_loop = real(countD1-countD0)/(count_rate)
```

```
#include <time.h>
…
time1 = clock();
// do something //
time2 = clock();
dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
```

Tools

Many tool are (or were) used to (also) to profile/monitoring a code and much more, for example

- ✓ GNU perf
- ✓ GNU gprof
- ✓ Intel vtune
- ✓ Nvidia nsight
- ✓ AMD μprof
- ✓ scalasca
- ✓ likwid
- ✓ ….

✓ Old but useful: gprof flat profile

....

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds  calls  ms/call  ms/call  name
 34.25   0.13      0.13   3628800   0.00     0.00    check1
 31.61   0.25      0.12  36288000   0.00     0.00    d
 10.54   0.29      0.04         1   40.04   40.04    getgeom
  9.22   0.33      0.04  12470600   0.00     0.00    swap
  7.90   0.36      0.03  72576000   0.00     0.00    sqr
  5.27   0.38      0.02         1   20.02   340.34   search1
  1.32   0.38      0.01   3628800   0.00     0.00    save
  0.00   0.38      0.00         1   0.00    340.34   solve1
```

gprof

✓ Old but useful: gprof call profile

```
index     %    time  self  children   called   name
                                                   <spontaneous>
[1]  100.0    0.00  0.38                          main [1]
              0.00  0.34    1/1                   solve1 [3]
              0.04  0.00    1/1                   getgeom [6]
----------------------------------------------
                            6235300                search1 [2]
              0.02  0.32       1/1                 solve1 [3]
[2]  89.5    0.02  0.32    1+6235300              search1 [2]
              0.13  0.16  3628800/3628800          check1 [4]
              0.04  0.00 12470600/12470600        swap [7]
                              6235300             search1 [2]
----------------------------------------------
```

`gprof` sintax

A three step procedure

1. compile (all) the code with option "**`-pg`**"
   - `gcc -pg myfile.c -o myfile.x`
2. run the exe: it will create a gmon.out binary file
3. process the gmon.out file (standard output)
   - `gprof ./myfile.x > gprof.txt`

Caveat

- ✓ I could be intrusive
- ✓ Also linking step with flag "**`-pg`**"
- ✓ (unfortunately) is no more supported by some compilers :-(

Profile: Best Practices

- ✓ Check with your code complexity which performance are you expecting
  - If they don’t agree there’s an issue
- ✓ Verify to be big enough not to fit entirely in cache
  - Verify performance for all code/function/subroutines
  - Cache hide real code latencies
- ✓ Take care of profiling intrusivity
  - Always take care of timing with or without profiling
- ✓ Use different HW/SW
  - Can help to detect “strange” behaviour
- ✓ Useful to understand the code flow
  - especially if you haven’t written the code!

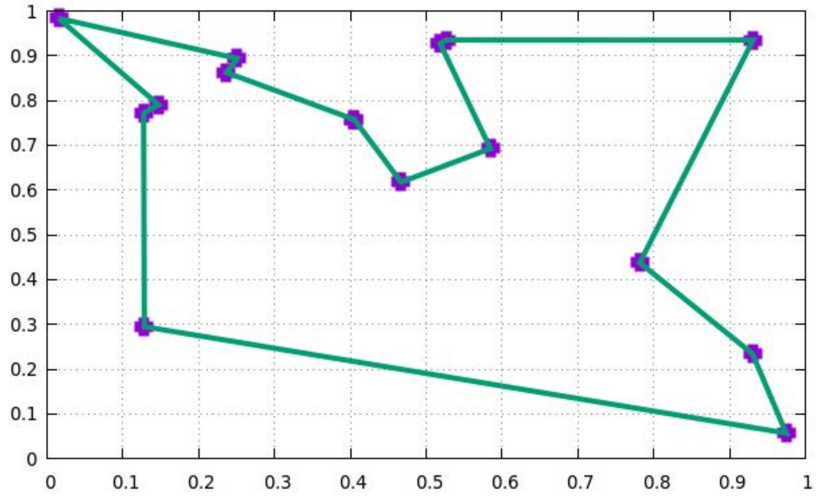
Example: Travel Salesman

https://en.wikipedia.org/wiki/Travelling_salesman_problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

It is NP-complex problem: our approach

1. Brute force
2. Reducing combinations
3. Reducing functions
4. Reducing comparison
5. Precomputing distances



Travel Salesman/1

Brute-force approach: compute al N! combinations, where N is the number of cities to visit, unfortunately

- ✓ 10! = 3628800
- ✓ 11! = 479001600
- ✓ …
- ✓ 20!= 2432902008176640000

Really unfeasible! Any Idea?

```
time    seconds    seconds  calls   ms/call  ms/call  name
 34.25   0.13      0.13    3628800    0.00     0.00   check1
 31.61   0.25      0.12   36288000    0.00     0.00   d
 10.54   0.29      0.04          1   40.04    40.04   getgeom
  9.22   0.33      0.04   12470600    0.00     0.00   swap
  7.90   0.36      0.03   72576000    0.00     0.00   sqr
  5.27   0.38      0.02          1   20.02   340.34    search1
…
```

Travel Salesman/2

Reducing combinations

- ✓ We don’t need to compute ALL the distancies. A lot are computed many times
- ✓ the distances with 4 cities (A,B,C,D)
  - ABCD = BCDA = CDAB = DABC
- ✓ You don’t have to compute N! distances but (only) (n+1)! starting from a fixed city
- ✓ Big gain as N increase

time	seconds	seconds	calls	ms/call	ms/call	name
66.73	0.02	0.02	362880	0.00	0.00	check1
33.37	0.03	0.01	3628800	0.00	0.00	d
0.00	0.03	0.00	7257600	0.00	0.00	sqr
0.00	0.03	0.00	1247058	0.00	0.00	swap
0.00	0.03	0.00	362880	0.00	0.00	save

Travel Salesman/3

Reducing functions

- ✓ We don't need to recompute all the distances for every single tour.
- ✓ Fix the distance between m cities and compute distance for remaining n-m cities

```
time    seconds    seconds      calls  ms/call  ms/call  name
 35.75      0.03       0.03   1349289     0.00     0.00  d
 21.45      0.04       0.02         1    15.02    65.07  search3
 14.30      0.05       0.01   1247058     0.00     0.00  swap
 14.30      0.06       0.01    362880     0.00     0.00  check3
  7.15      0.07       0.01   2698578     0.00     0.00  sqr
```

Travel Salesman/4

Check distances

✓ instead of checking the final distance of a tour for the minimal one, stop the search if the (partial) tour is longer of the minimum found

```
time   seconds   seconds  calls  ms/call  ms/call  name
100.10   0.01     0.01   111515    0.00     0.00  d
  0.00   0.01     0.00   223030    0.00     0.00  sqr
  0.00   0.01     0.00   213374    0.00     0.00  swap
  0.00   0.01     0.00     2414    0.00     0.00  check3
  0.00   0.01     0.00     2414    0.00     0.00  save
```

Travel Salesman/5

Precomputing distancies

- ✓ The distance between A and B will be computed many time (i.e. build a **look-up table**)
- ✓ Precompute once and then access (matrix of n^2 elements)

time	seconds	seconds	calls	ms/call	ms/call	name
100.10	0.01	0.01	1	10.01	10.01	search5
0.00	0.01	0.00	213374	0.00	0.00	swap
0.00	0.01	0.00	2414	0.00	0.00	check5
0.00	0.01	0.00	2414	0.00	0.00	save
0.00	0.01	0.00	200	0.00	0.00	sqr

Travel Salesman: some figures

Results, in seconds, using IBM Power4@1100 Mhz (many many time ago)

	9	10	11	12	13	14	15
1	0.92	10.3	123	1538	-	-	-
2	0.10	1.03	11.2	134	-	-	-
3	-	0.46	4.59	50.5	606	-	-
4	-	-	0.29	1.50	11.3	98.7	-
5	-	-	0.11	0.57	4.29	37.6	288

- ✓ Reducing unnecessary operation
- ✓ Removing repetition
- ✓ profiling help to focus on function to remove!

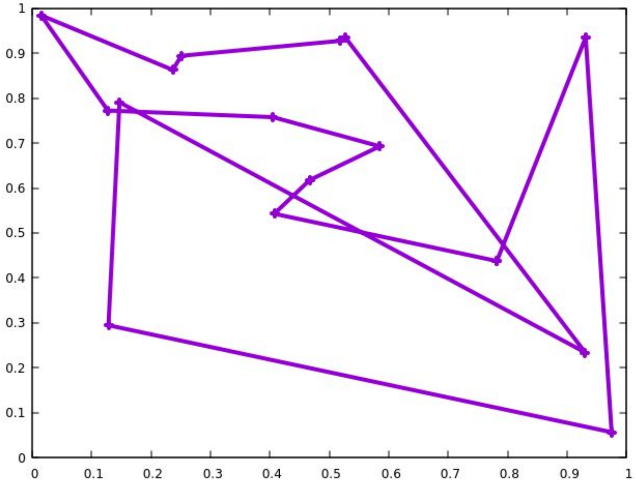
Travel Salesman: some (new) figures

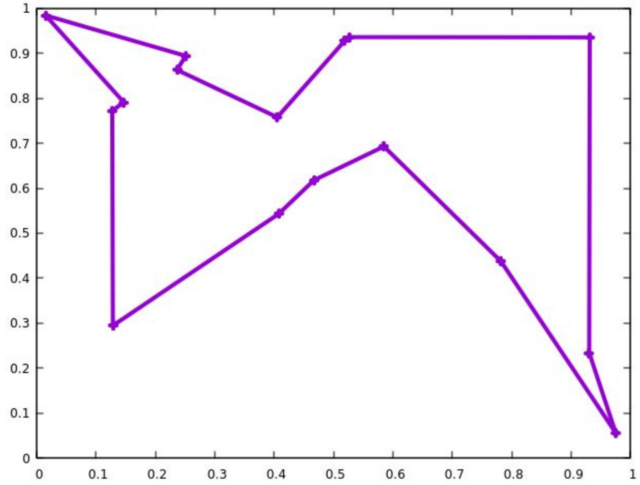
Results, in seconds, using the AMD Ryzen 5 5625U (with gcc)

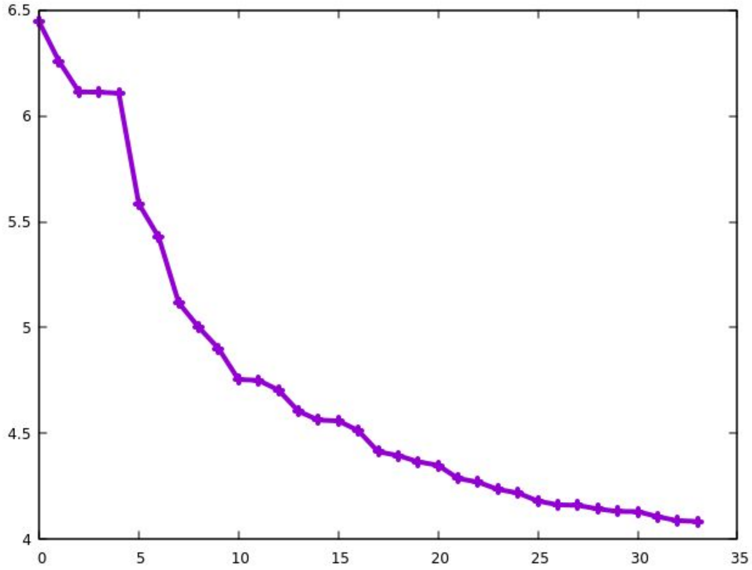
	13	14	15	16
4	0.66”	5.75”	43.6”	-
5	0.39”	3.33”	24.6”	170”
5 - initial condition	-	-	-	24.7”

- ✓ Reducing unnecessary operation
- ✓ Removing repetition
- ✓ a good initial condition can help (from rel. 4)
- ✓ profiling help to focus on operations to remove/optimize!
- ✓ HW improvements helps, but SW improvements are larger!!!

Travel Salesman: 16 cities

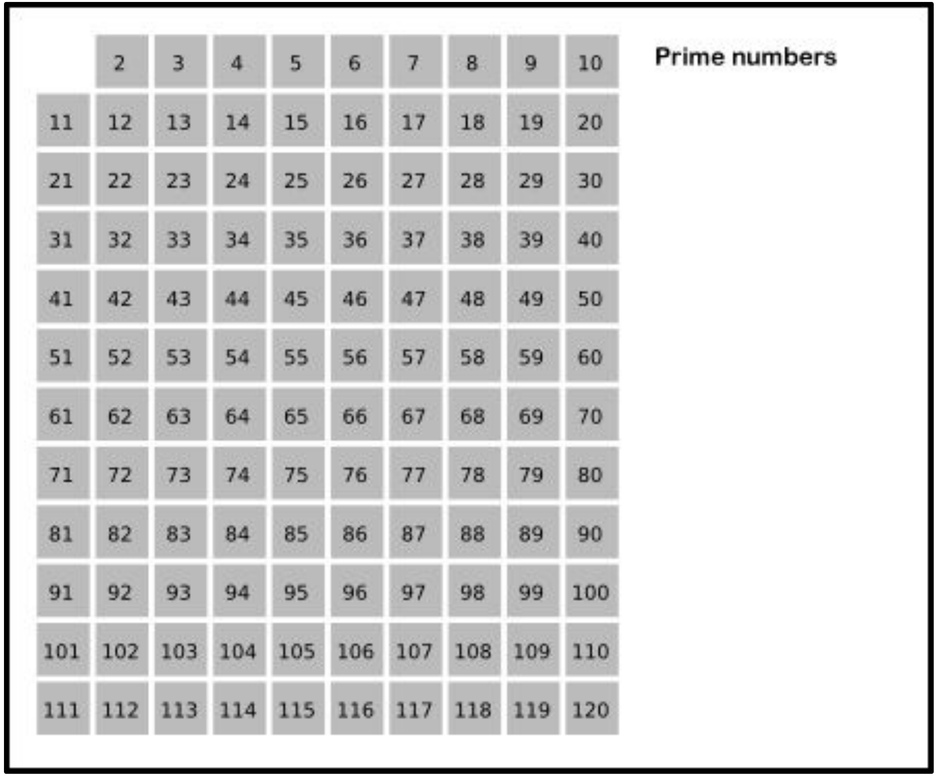






Sieve of Eratosthenes

- ✓ https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- ✓ One of the first known Algorithm for finding prime numbers
  - ■ much older than computers!
  - ■ It is an iterative algorithm



Sieve of Eratosthenes: step0

- ✓ https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- ✓ First implementation

```
parameter(nd=50000)                 ! max number to check
…
do i = 2, nd
   a(i) = 0                         ! possible prime number
enddo
…
do icheck = 2, nd
     do i = icheck+1, nd
          if (mod(i,icheck).eq.0) then
              a(i) = 1              ! i is not a prime number
          endif
     enddo
enddo
```

Sieve of Eratosthenes: figures

✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-

Sieve of Eratosthenes: step 1

- ✓ The original code performs a lot of “useless” checks.
- ✓ It checks, for example, all the numbers that are divisible by 4, that are a subset of those divisible by 2.
- ✓ You have to check only with numbers that you know are candidate as prime, i.e. those with a(i)=0

```
parameter(nd=50000)              ! max number to check
…
do icheck = 2, nd
   do i = icheck+1, nd
      if (a(i).eq.0) then
          if (mod(i,icheck).eq.0) then
              a(i) = 1           ! i is not a prime number
          endif
      endif
   enddo
enddo
```

Sieve of Eratosthenes: figures

✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-
Step1	2383	53.5%

Sieve of Eratosthenes: step 2

- ✓ Other “useless” operations to skip
- ✓ If you know that a number is not a prime you can skip the check

```
parameter(nd=50000)                   ! max number to check
…
do icheck = 2, nd
   if (a(icheck).eq.0) then
      do i = icheck+1, nd
         if (a(i).eq.0) then
             if (mod(i,icheck).eq.0) then
                 a(i) = 1             ! i is not a prime number
             endif
         endif
      enddo
   endif
enddo
```

Sieve of Eratosthenes: figures

✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-
Step1	2383	53.5%
Step2	212	95.8%

Sieve of Eratosthenes: step 3

- ✓ Other “useless” operations to skip
- ✓ If m is the biggest prime number found you must start the search from m*m

```
parameter(nd=50000)                    ! max number to check
…
do icheck = 2, nd
   if (a(icheck).eq.0) then
      do i = icheck*icheck, nd
         if (a(i).eq.0) then
             if (mod(i,icheck).eq.0) then
                 a(i) = 1         ! i is not a prime number
             endif
         endif
      enddo
   endif
enddo
```

Sieve of Eratosthenes: figures

✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-
Step1	2383	53.5%
Step2	212	95.8%
Step3	1.21	99.976%

Sieve of Eratosthenes: step 4

✓ Instead of checking modulo (i.e. a division) we mark directly the multiple (i.e. multiplication)

```
parameter(nd=50000)                     ! max number to check
…
do icheck = 2, nd
   if (a(icheck).eq.0) then
      imax = INT(nd/i)
      do i=1,imax
         a(i*icheck) = 1                ! i is not a prime number
      enddo
   endif
enddo
```

Sieve of Eratosthenes: figures

✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-
Step1	2383	53.5%
Step2	212	95.8%
Step3	1.21	99.976%
Step4	0.13	99.9974%

Sieve of Eratosthenes: step 5 (fine tuning)

- ✓ Mark as non-prime number up to 30 in the initialization step
- ✓ Where is the problem?

```
a( 2) = 0                    ! prime
a( 3) = 0                    ! prime
a( 4) = 1
a( 5) = 0                    ! prime
....
a(30) = 1
do i=30,nd,30
   a(i+ 1) = 0               ! could be a prime
   a(i+ 2) = 1
   a(i+ 3) = 1
   a(i+ 4) = 1
   a(i+ 5) = 1
    ....
   a(i+29) = 0                ! could be a prime
   a(i+30) = 1
enddo
```

Sieve of Eratosthenes: figures

✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-
Step1	2383	53.5%
Step2	212	95.8%
Step3	1.21	99.976%
Step4	0.13	99.9974%
Step5	0.094	99.9982%

Sieve of Eratosthenes: step 6 (fine tuning)

- ✓ Limit the search up to sqrt(nd)
  - ■ it could be is the biggest prime number up to nd

```
parameter(nd=50000)                    ! max number to check
…
do icheck = 7, INT(sqrt(float(nd))
   if (a(icheck).eq.0) then
      imax = INT(nd/i)
      do i=1,imax
         a(i*icheck) = 1               ! i is not a prime number
      enddo
   endif
enddo
```

Sieve of Eratosthenes: figures

- ✓ Intel PentiumIII, 700 Mhz (old stuff)

	Time (s)	Gain (%)
Step0	5118	-
Step1	2383	53.5%
Step2	212	95.8%
Step3	1.21	99.976%
Step4	0.13	99.9974%
Step5	0.094	99.9982%
Step6	0.078	99.9985%

Instructions vs. cycles

- ✓ For a CPU some instructions are expensive
- ✓ Other could be less expensive
  - Data access
  - mathematical functions
    - (old measurements)

Function	Cycles
sin(x)	229
exp(x)	535
log(x)	423
acos(x)	634
x**1.5	945
x**2	26.5
x**2.0	675
sqrt(x)	53
1/x	50

Different operations, different cost

- ✓ Different costs
  - sum → few cycles
  - product → few cycles
  - division → many cycles
  - square root → many many cycles
  - exponent → many cycles
  - trigonometric functions → many cycles
  - data access → few/many access

- ✓ (Possible) Solutions
  - Lookup table
  - If range limited
    - fast libraries
    - polynomial approximation
    - taylor series
    - prostaferesi formula

Instructions vs. cycles

- ✓ Mathematical functions can be really expensive
  - https://linasm.sourceforge.net/docs/api/math.php
  - via compiler (e.g. `-ffast-math`)
  - `-lmaas` (IBM)

- ✓ Using intel vml (vectorized mathematical library)
- ✓ (old stuff)

Function	Cycles	vml
sin(x)	229	54
exp(x)	535	32
log(x)	423	50
acos(x)	634	59
x**1.5	945	225
x**2	26.5	27
x**2.0	675	661
sqrt(x)	53	18
1/x	50	27

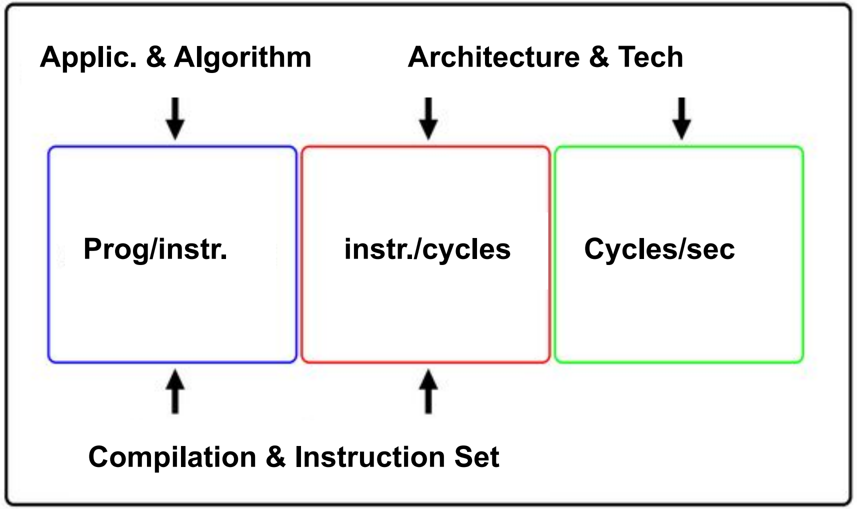
Comments

- ✓ Profiling could help
  - find the most expensive functions
  - HW counters can give some feedbacks about cycles per instruction
- ✓ Know your enemy!
  - Are the function computed for same values?
    - pre-compute once
  - Are the function computed for similar values?
    - approximated solution
    - taylor approximation

Who is in charge of what (reprise)?

High performance is the joint effort of different “players”

- ✓ **Programmer**: in charge of the choice of the algorithm
- ✓ **Compiler**: in charge of the “language” traduction in instructions
- ✓ **HW**: in charge of executing the instructions



(some) Keywords

✓ **Look-up table**: precomputed data. It could be a gain in time (size dependent)
✓ **Roofline model**
✓ **Algorithm Intensity**
✓ **Bandwidth limited code**
✓ **FP limited code**

Take home message

- ✓ Take care of the chosen Algorithm
  - You (should) know how it works and which operations perform or remove (e.g. TSP)
- ✓ Take care of Algorithm Implementation
  - Data allocation
  - Data access
  - Operation to be performed
  - Solution of a linear system
- ✓ timing/profiling could help
  - in finding bottlenecks
  - in understanding where you’re wrong
- ✓ In general: coding style could seriously affect performance
  - Loop ordering
- ✓ Some operation are more “expensive” with respect to others

Some references

- ✓ [Bentley, More Programming Pearls (2nd Ed.), ACM Press/Addison-Wesley, 1989]
- ✓ [Bentley, DDJ, 1999]
- ✓ [Discovering faster matrix multiplication algorithms with reinforcement learning, Nature, 2023],
- ✓ [Dispense Master HPC 2007]