Introduction to High-Performance Computing

Giorgio Amati
Alessandro Ceci

Corso di dottorato in Ingegneria Aeronautica e Spaziale 2025

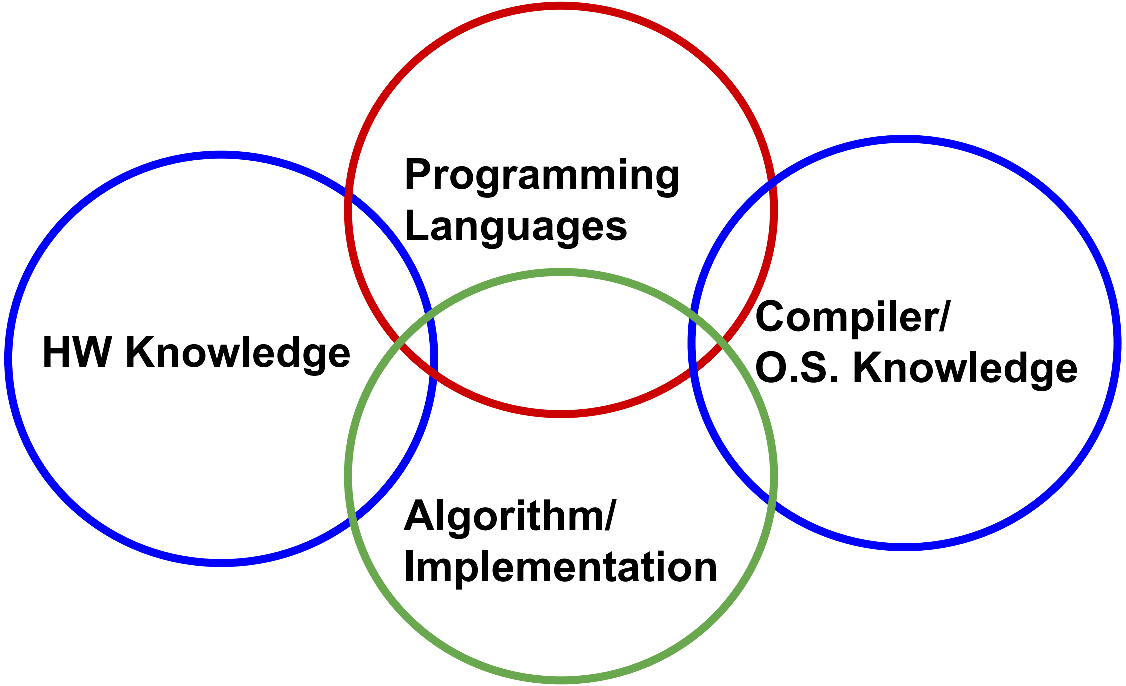
g.amati@cineca.it/g.amaticode@gmail.com
alessandro.ceci@uniroma1.it

Agenda

- ✓ **HPC: What is it?**
- ✓ **Hardware: how it works**
- ✓ **Algorithm vs. Implementation**
- ✓ **Compiler + Floating point + I/O**
- ✓ Parallel Paradigm
- ✓ Conclusions & Comments

HPC: what it is?

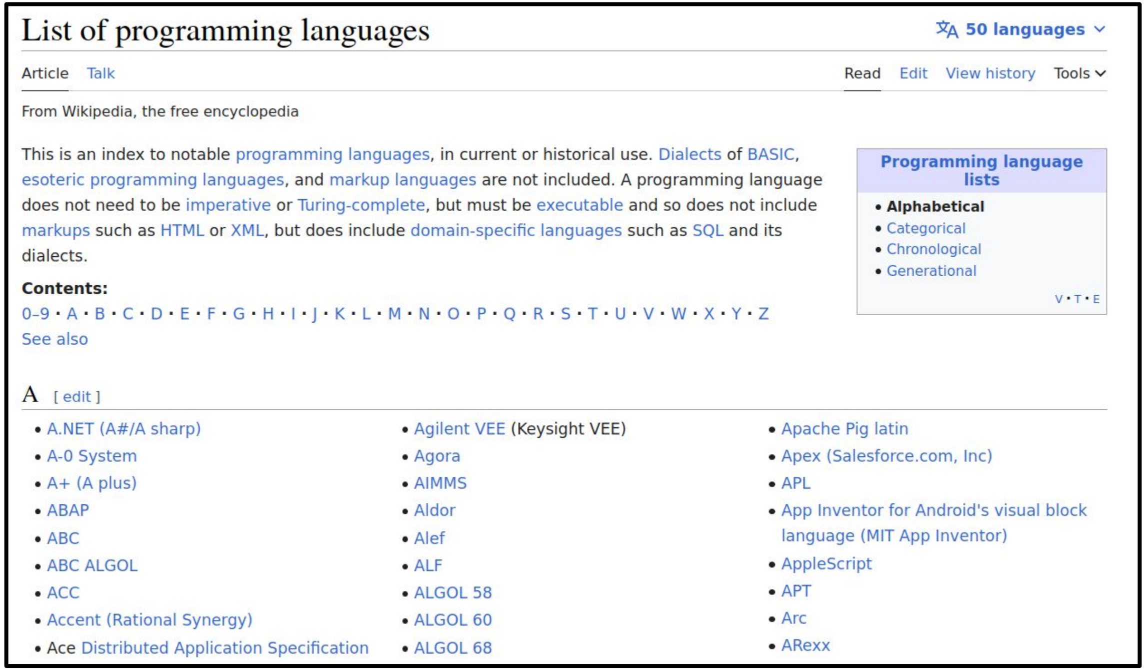
✓ These are the main skills for an efficient HPC



Which programming language?

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

✓ 690 different programming languages
  - TeX
  - Postscript
  - C- -
  - Ballerina
  - Mystic Programming Language (MPL)



List of programming languages

文A 50 languages

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

This is an index to notable programming languages, in current or historical use. Dialects of BASIC, esoteric programming languages, and markup languages are not included. A programming language does not need to be imperative or Turing-complete, but must be executable and so does not include markups such as HTML or XML, but does include domain-specific languages such as SQL and its dialects.

Programming language lists
- **Alphabetical**
- Categorical
- Chronological
- Generational

V • T • E

Contents:
0–9 • A • B • C • D • E • F • G • H • I • J • K • L • M • N • O • P • Q • R • S • T • U • V • W • X • Y • Z
See also

A [edit]

- A.NET (A#/A sharp)
- A-0 System
- A+ (A plus)
- ABAP
- ABC
- ABC ALGOL
- ACC
- Accent (Rational Synergy)
- Ace Distributed Application Specification
- Agilent VEE (Keysight VEE)
- Agora
- AIMMS
- Aldor
- Alef
- ALF
- ALGOL 58
- ALGOL 60
- ALGOL 68
- Apache Pig latin
- Apex (Salesforce.com, Inc)
- APL
- App Inventor for Android's visual block language (MIT App Inventor)
- AppleScript
- APT
- Arc
- ARexx

Programming Languages

- ✓ A **compiled language** is a programming language whose implementations pass through a compiler. It is a translator that generates machine code from source code
  - e.g. Fortran, C, C++, …
  - Syntax must be correct before running
  - Can be optimized by the compiler
- ✓ An **interpreted language** is a programming language that is executed step-by-step, with no pre-runtime translation
  - E.g. Perl, Python, bash, ….
  - It runs until it finds a syntax error
  - Optimization usually could be very low

Compiler: how it works?

compiler *is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a low-level programming language (e.g. assembly language, object code, or machine code) to create an executable program*

1. **Front-End**
   a. Pre-processing
   b. Syntax & Semantic check/analysis
2. **Middle-End**
   a. (general) Optimization
3. **Back-End**
   a. Machine dependent optimization
   b. Code Generation

Compiler: Optimization level

Have you ever read compiler's manual?

What the different level of optimization means? For `nvfortran` compiler

- ✓ **-O:** All -O1 optimizations plus traditional global scalar optimizations performed
- ✓ **-O0**: Creates a basic block for each statement. No scheduling or global optimizations performed
- ✓ **-O1**: Some scheduling and register allocation is enabled. No global optimizations performed
- ✓ **-O2**: All -O optimizations plus SIMD code generation. Implies -Mvect=simd
- ✓ **-O3**: All -O2 optimizations plus more aggressive code hoisting and scalar replacement, that may or may not be profitable, Implies -Mvect=simd
- ✓ **-O4**: All -O3 optimizations plus more aggressive hoisting of guarded expressions performed. Implies -Mvect=simd
- ✓ **-fast**: Common optimizations; includes -O2 -Munroll=c:1 -Mlre -Mautoinline, Implies -Mvect=simd -Mflushz -Mcache_align

Optimization level example: MMM

✓ Different level of optimization for size=1000 matrices (on different machine)

Opzione	IBM xlf (sec.)	HP f77 (sec.)	Portland pgf77 (sec.)
-O0	176	185	32.0
-O1	-	237	30.9
-O2	121	115	29.6
-O3	122	98.1	-
-O4	3.1	98.4	-
-O5	3.8	4.9	-

MMM

- ✓ Old example (IBM)
- ✓ Old but “quite” clear
- ✓ Where does this difference in time comes from?

Option	Time (sec.)
-O0	24”
-O2	6.35”
-O3	4.87”
-O4	2.14”

Reading the assembler

✓ Old example (IBM)

```
do k = 1, n
   do j = 1, n
      do i = 1, n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
   enddo
endd0
```

✓ Compiler can produce, if you activate the right flags

- Listing file (for IBM *.lst)
- Assembler file (for IBM *.s)

Recap: load/store architecture

reading the assembler: -O0 option

- ✓ mm.lst

```
68| 0004B4   lwz      80820008   1   L4A    gr4=.$STATIC_BSS(gr2,0)
68| 0004B8   lwz      80640004   2   L4A    gr3=j(gr4,4)
68| 0004BC   rlwinm   54636824   2   SLL4   r3=gr3,13
68| 0004C0   lwz      80A40008   1   L4A    gr5=i(gr4,8)
68| 0004C4   rlwinm   54A51838   2   SLL4   gr5=gr5,3
68| 0004C8   add      7CC32A14   1   A      gr6=gr3,gr5
68| 0004CC   addis    3CE40100   1   AIU    gr7=gr4,256
68| 0004D0   add      7CC63A14   1   A      gr6=gr6,gr7
68| 0004D4   lfd      C826E018   1   LFL    fp1=c[](gr6,-8168)
68| 0004D8   lwz      80C40010   1   L4A    gr6=k(gr4,16)
68| 0004DC   rlwinm   54C76824   2   SLL4   gr7=gr6,13
68| 0004E0   add      7CE53A14   1   A      gr7=gr5,gr7
68| 0004E4   add      7CE43A14   1   A      gr7=gr4,gr7
68| 0004E8   lfd      C847E018   1   LFL    fp2=a[](gr7,-8168)
…
```

reading the assembler: -O0 option

```
…
68│ 0004EC   rlwinm   54C61838   1   SLL4   gr6=gr6,3
68│ 0004F0   add      7CE33214   1   A      gr7=gr3,gr6
68│ 0004F4   addis    3CC40080   1   AIU    gr6=gr4,128
68│ 0004F8   add      7CC63A14   1   A      gr6=gr6,gr7
68│ 0004FC   lfd      C866E018   1   LFL    fp3=b[](gr6,-8168)
68│ 000500   fmadd    FC2208FA   1   FMA    fp1=fp1-fp3,fcr
68│ 000504   add      7C632A14   0   A      gr3=gr3,gr5
68│ 000508   addis    3C840100   1   AIU    gr4=gr4,256
68│ 00050C   add      7C632214   1   A      gr3=gr3,gr4
68│ 000510   stfd     D823E018   1   STFL   c[](gr3,-8168)=fp1
```

✓ mm.lst: 24 instructions but

- only 1 store
- only 3 load
- only 1 fused-multiply and add (i.e. 2 flops)
- → 2/24 peak performance (~8%)

reading the assembler: -O0 option

- ✓ mm.lst: 24 instructions but
  - only 1 store
  - only 3 load
  - only 1 fused-multiply and add (i.e. 2 flops)
  -
- ✓ Other 19 instructions where do they come from?

- ✓ Integer operation to compute address (i.e. element location in memory)
- ✓ 19/24 of total instructions “spent” in understanding where data is!!!
- ✓ ~80% of total time only to “understand”, data fetching could be more expensive (e.g. cache miss)

reading the assembler: -O2 option

- ✓ mm.lst

```
68|      0003D8    lfd     C8070008     1     LFL     fp0=c(gr7,8)
 0|      0003DC    mtspr   7F8903A6     1     LCTR    ctr=gr28
 0|      0003E0    ori     63080000     1     LR      gr8=gr24
 0|      0003E4    lfdu    CC662000     1     LFDU    fp3,gr6=b(gr6,8192)
 0|      0003E8    ori     60E90000     1     LR      gr9=gr7
68|      0003EC    lfdu    CC280008     1     LFDU    fp1,gr8=a(gr8,8)
68|      0003F0    fmadd   FC0100FA     1     FMA     fp0=fp0,fp1,fp3,fcr
 0|      0003F4    bc      43400018     0     BCF     ctr=CL.111,taken=0%(0,100)
68|      0003F8    lfd     C8490010     1     LFL     fp2=c(gr9,16)
68|      0003FC    lfdu    CC280008     1     LFDU    fp1,gr8=a(gr8,8)
68|      000400    stfdu   DC090008     1     STFDU   gr9,c(gr9,8)=fp0
68|      000404    fmadd   FC0110FA     1     FMA     fp0=fp2,fp1,fp3,fcr
 0|      000408    bc      4320FFF0     0     BCT     ctr=CL.29,taken=100%(100,0)
 0|                                        CL.111:
68|      000410    stfdu   DC09000      1     STFDU    gr9,c(gr9,8)=fp0
…
```

reading the assembler: -O2 option

- ✓ mm.lst: 15 instructions
  - 2 store
  - 5 load
  - 2 fused-multiply and add
  - → 4/15 peak performance (~26%)

- ✓ mm.s

```
__L3d8:                                        # 0x000003d8 (H.10.NO_SYMBOL+0x3d8)
        lfd     fp0,8(r7)
        mtspr   CTR,r28
        oril    r8,r24,0x0000
        lfdu    fp3,8192(r6)
        oril    r9,r7,0x0000
        lfdu    fp1,8(r8)
        fma     fp0,fp1,fp3,fp0
        .machine        "any"
        bc      BO_dCTR_ZERO_8,CR0_LT,__L40c
__L3f8:                                        # 0x000003f8 (H.10.NO_SYMBOL+0x3f8)
        lfd     fp2,16(r9)
        lfdu    fp1,8(r8)
        stfdu   fp0,8(r9)
        fma     fp0,fp1,fp3,fp2
        bc      BO_dCTR_NZERO_9,CR0_LT,__L3f8
```

reading the assembler: -O3 option

✓ mm.lst

```
68|      000478   lfd    C8910008   1    LFL    fp4=a(gr17,8)
 0|      00047C   mtspr  7D0903A6   1    LCTR   ctr=gr8
 0|      000480   ori    62260000   1    LR     gr6=gr17
 0|      000484   lfdu   CC042000   1    LFDU   fp0,gr4=b(gr4,8192)
 0|      000488   ori    60A70000   1    LR     gr7=gr5
68|      00048C   lfd    C8650008   1    LFL    fp3=c(gr5,8)
68|      000490   lfd    C8D10010   1    LFL    fp6=a(gr17,16)
68|      000494   lfd    C8E50010   1    LFL    fp7=c(gr5,16)
68|      000498   lfd    C8310018   1    LFL    fp1=a(gr17,24)
68|      00049C   lfdu   CCA60020   1    LFDU   fp5,gr6=a(gr6,32)
68|      0004A0   lfd    C8450018   1    LFL    fp2=c(gr5,24)
68|      0004A4   fmadd  FC64183A   1    FMA    fp3=fp3,fp4,fp0,fcr
68|      0004A8   lfd    C8850020   1    LFL    fp4=c(gr5,32)
68|      0004AC   fmadd  FCC6383A   1    FMA    fp6=fp7,fp6,fp0,fcr
68|      0004B0   bc     43400048   0    BCF    ctr=CL.126,taken=0%(0,100)
68|                   CL.127:
68|      0004B4   fmadd  FC21103A   1    FMA    fp1=fp2,fp1,fp0,fcr
```

reading the assembler: -O3 option

✓ mm.lst

```
68|      0004B8   stfd   D8670008    1   STFL    c(gr7,8)=fp3
68|      0004BC   lfd    C8470028    1   LFL     fp2=c(gr7,40)
68|      0004C0   stfd   D8C70010    1   STFL    c(gr7,16)=fp6
68|      0004C4   lfd    C8660008    1   LFL     fp3=a(gr6,8)
68|      0004C8   fmadd  FC85203A    1   FMA     fp4=fp4,fp5,fp0,fcr
68|      0004CC   lfd    C8A70030    1   LFL     fp5=c(gr7,48)
68|      0004D0   lfd    C8C60010    1   LFL     fp6=a(gr6,16)
68|      0004D4   stfd   D8270018    1   STFL    c(gr7,24)=fp1
68|      0004D8   fmadd  FC63103A    1   FMA     fp3=fp2,fp3,fp0,fcr
68|      0004DC   lfd    C8470038    1   LFL     fp2=c(gr7,56)
68|      0004E0   lfd    C8260018    1   LFL     fp1=a(gr6,24)
68|      0004E4   fmadd  FCC6283A    1   FMA     fp6=fp5,fp6,fp0,fcr
68|      0004E8   stfdu  DC870020    1   STFDU   gr7,c(gr7,32)=fp4
68|      0004EC   lfd    C8870020    1   LFL     fp4=c(gr7,32)
68|      0004F0   lfdu   CCA60020    1   LFDU    fp5,gr6=a(gr6,32)
 0|      0004F4   bc     4320FFC0    0   BCT     ctr=CL.127,taken=100%(100,0)
68|                   CL.126:
```

reading the assembler: -O3 option

- ✓ mm.lst: 34 instructions
  - 4 store
  - 16 load
  - 6 fused-multiply and add
  - → 12/34 peak performance (~35%)
- ✓ mm.s

```
lfd     fp4,8(r17)
mtspr   CTR,r8
oril    r6,r17,0x0000
lfdu    fp0,8192(r4)
oril    r7,r5,0x0000
lfd     fp3,8(r5)
lfd     fp6,16(r17)
lfd     fp7,16(r5)
lfd     fp1,24(r17)
lfdu    fp5,32(r6)
lfd     fp2,24(r5)
fma     fp3,fp4,fp0,fp3
lfd     fp4,32(r5)
fma     fp6,fp6,fp0,fp7
```

reading the assembler: -O5 option

- ✓ mm.lst: 344 instructions
  - 64 store
  - 128 load
  - 100 fused-multiply and add
  - → 200/344 peak performance (~58%)
  - → unrolling outer loop
  - → blocking

MMM

- ✓ Old example (IBM pwr)
- ✓ Old but quite clear
- ✓ Now we know where these figures comes from

Option	Time (sec.)	%flops instructions
-O0	24”	8%
-O2	6.35”	26%
-O3	4.87”	37%
-O4	2.14”	58%

Compiler: what it can do?

Many optimization:

- ✓ Dead and redundant code removal
- ✓ Copy propagation
- ✓ Code motion
- ✓ Strength reduction
- ✓ Common subexpression elimination
- ✓ Register allocation
- ✓ Loop pipelining/unrolling
- ✓ Cache blocking
- ✓ Loop interchange/reordering
- ✓ Function inlining
- ✓ ….

Code Motion

✓ A compiler can move instructions (e.g. outside of a loop) only if it realizes that the result is not affected by this transformation

```
do i = 1, n
   somma = somma + a(i)*b(j)*c(k)
enddo
```

```
do i = 1, n
   somma = somma + (a(i)*b(j))*c(k)
enddo
```

```
temp = b(j)*c(k)
do i = 1, n
   somma = somma + a(i)*temp
enddo
```

Common Subexpression Elimination

✓ A compiler can group operations to reduce total number of instructions

```
x = c*sin(a)*cos(b)
y = c*sin(a)*sin(b)
x = c*cos(a)
```

```
temp = c*sin(a)
x = temp*cos(b)
y = temp*sin(b)
z = c*cos(a)
```

Strength Reduction

- ✓ Math Functions can be very expensive
- ✓ the compiler can change one operation with another equivalent one but less expensive
  - ■ a**2 → a*a
- ✓ but only at high level of optimization
- ✓ or use fast math libraries
  - ■ but result can be different → Finite precision

Function	Cycles
sin(x)	230
exp(x)	535
log(x)	423
acos(x)	635
x**1.5	945
x**2	27
sqrt(x)	54
x**2.0	674
1/x	50

inlining

***inline expansion**, or **inlining**, is a manual or compiler optimization that replaces a function call site with the body of the called function. Inline expansion is similar to macro expansion, but occurs during compilation, without changing the source code (the text), while macro expansion occurs prior to compilation, and results in different text that is then processed by the compiler.*

- ✓ The code is less compact and more verbose
- ✓ All overhead of function is removed, allowing further optimization
- ✓ profiling can help to spot fast functions
  - → `gprof`

inlining: example

- ✓ The code is less compact and more verbose
- ✓ Overheads of function are removed, allowing further optimization (e.g. unrolling)

```
do i = 1, n
   ......
   call somma(a,b,c)
   ......
enddo

subroutine somma
real a,b,c
c = c + a*b
return
```

```
do i = 1, n
   ......
   c = c + a*b
   ......
enddo
```

Loop reordering

- ✓ Using high optimization levels, the compiler “can” reorder loops to make it more efficient (i.e. cache efficient)
- ✓ But it can be confused by
  - Coding style
  - pointers
  - functions inside the loop
  - too much nested loops
  - I/O statements

```
do i = 1, n
    do k = 1, n
        do j = 1, n
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
        enddo
    enddo
enddo
```

Loop reordering: example

✓ 2048 Matrix size: 114'' (-O0) vs. 3.3'' (-O3)

```
gamati01@dgx03:/$ nvfortran -O0 -Minfo mm.F90 mod_tools.F90  -o mm.slow
mm.F90:
mm:
     75, FMA (fused multiply-add) instruction(s) generated
```

```
gamati01@dgx03:/$ nvfortran -O3 -Minfo mm.F90 mod_tools.F90 -o mm.fast
mm.F90:
mm:
     64, Memory zero idiom, array assignment replaced by call to pgf90_mzero8
     72, Loop interchange produces reordered loop nest: 74,73,72
         Generated vector simd code for the loop
     73, Zero trip check eliminated
     74, Zero trip check eliminated
         Loop not fused: function call before adjacent loop
     75, FMA (fused multiply-add) instruction(s) generated
```

Loop reordering

- ✓ Too much loops can confuse the compiler
- ✓ Remember: compilers are “conservative”. They do not perform optimization unless they are sure of having no side effects!

```
do ii = 1, n, step
   do kk = 1, n, step
      do jj = 1, n, step
         do j = jj, jj+step-1
            do k = kk, kk+step-1
               do i = ii, ii+step-1
                  c(i,j) = c(i,j) + a(i,k)*b(k,j)
               enddo
            enddo
         enddo
      enddo
   enddo
enddo
```

Loop reordering: example

✓ 4096 Matrix size: 384” (-O0) vs. 19.1” (-O3) vs 11.1” (-O3, correct loop ordering)

```
gamati01@dgx03:/$ nvfortran -O0 -Minfo mm.F90 mod_tools.F90 -o mm.slow
mm:
     75, FMA (fused multiply-add) instruction(s) generated
```

```
gamati01@dgx03:$ nvfortran -O3 -Minfo mm.F90 mod_tools.F90 -o mm.fast
mm:
     73, Memory zero idiom, array assignment replaced by call to pgf90_mzero8
     81, Loop not fused: function call before adjacent loop
         Loop not vectorized/parallelized: too deeply nested
     82, Loop not vectorized/parallelized: too deeply nested
     83, Loop not vectorized/parallelized: too deeply nested
     85, Zero trip check eliminated
     86, Zero trip check eliminated
         Generated vector simd code for the loop
     87, FMA (fused multiply-add) instruction(s) generated
```

pointers: an example

Kinetic code

✓ Using pointer to “swap” arrays: current → old

```
     real(mystorage), dimension(:,:,:), pointer :: a01,a02,a03,a04,a05
     real(mystorage), dimension(:,:,:), pointer :: a06,a07,a08,a09,a10
     real(mystorage), dimension(:,:,:), pointer :: a11,a12,a13,a14,a15
     real(mystorage), dimension(:,:,:), pointer :: a16,a17,a18,a19
!
     real(mystorage), dimension(:,:,:), pointer :: b01,b02,b03,b04,b05
     real(mystorage), dimension(:,:,:), pointer :: b06,b07,b08,b09,b10
     real(mystorage), dimension(:,:,:), pointer :: b11,b12,b13,b14,b15
     real(mystorage), dimension(:,:,:), pointer :: b16,b17,b18,b19
```

pointers: an example

```
nvfortran -DPGI  -O2 -Mnodepchk -DFUSED  -c col_MC.F90
…
 Mean   time  0.800468E-01    100/   20000
 Mean   time  0.811069E-01    200/   20000
 Mean   time  0.819973E-01    300/   20000
 Mean   time  0.813246E-01    400/   20000
 Mean   time  0.807486E-01    500/   20000
…
```

```
nvfortran -DPGI  -O2 -Mnodepchk -Mcontiguous -DFUSED  -c col_MC.F90
…
 Mean   time  0.426128E-01    100/   20000
 Mean   time  0.425005E-01    200/   20000
 Mean   time  0.423607E-01    300/   20000
 Mean   time  0.424811E-01    400/   20000
 Mean   time  0.421280E-01    500/   20000
…
```

pointers: language can help!

✓ Fortran2008 has an attribute for contiguous pointers to leverage compiler's efforts.

```
real(mystorage), dimension(:,:,:), contiguous, pointer :: a01,a02,a03,a04,a05
real(mystorage), dimension(:,:,:), contiguous, pointer :: a06,a07,a08,a09,a10
real(mystorage), dimension(:,:,:), contiguous, pointer :: a11,a12,a13,a14,a15
real(mystorage), dimension(:,:,:), contiguous, pointer :: a16,a17,a18,a19
```

```
nvfortran -DPGI  -O2 -Mnodepchk -DFUSED -c col_MC.F90
…
 Mean   time  0.430257E-01    100/   20000
 Mean   time  0.431001E-01    200/   20000
 Mean   time  0.431249E-01    300/   20000
 Mean   time  0.431014E-01    400/   20000
 Mean   time  0.433579E-01    500/   20000
…
```

Another example.....

Implementing Turbulence model (Smagorinsky Model) in a 3D Lattice Boltzmann model

$$\nu_{total} = \nu_0 + \underbrace{(C_S\Delta)^2|S|}_{\nu_t}$$

$$S_{ij} = -\frac{3\omega_s}{2\rho_{(0)}}\Pi_{ij}^{(neq)}$$

$$\Pi_{ij}^{(neq)} = \sum_q c_{qi}c_{qj}\, f_q^{(neq)}$$

An example…..

$$\Pi_{ij}^{(neq)} = \sum_q c_{qi} c_{qj} \, f_q^{(neq)}$$

- ✓ 19*6*2 multiplications
- ✓ 19*6 sums
- ✓ 19*6*3 loads

What can a compiler do?

```
do i
   do j
      do k
…
         do pop=1,19
            Pxx = Pxx + cx(pop)*cx(pop)*neq(pop)
            Pyy = Pyy + cy(pop)*cy(pop)*neq(pop)
            Pzz = Pzz + cz(pop)*cz(pop)*neq(pop)
            Pxy = Pxy + cx(pop)*cy(pop)*neq(pop)
            Pxz = Pxz + cx(pop)*cz(pop)*neq(pop)
            Pyz = Pyz + cy(pop)*cz(pop)*neq(pop)
         enddo
…
```

first "optimization"

$$\Pi_{ij}^{(neq)} = \sum_q c_{qi} c_{qj} \, f_q^{(neq)}$$

- ✓ 19*6*2 multiplications
- ✓ 19*6 sums
- ✓ 19*6*2 loads

What can a compiler do?
Maybe this…..

```
…
   Pxx = cx(01)*cx(01)*n01 + cx(02)*cx(02)*n02 + &
         cx(03)*cx(03)*n03 + cx(04)*cx(04)*n04 + &
         cx(05)*cx(05)*n05 + cx(06)*cx(06)*n06 + &
         cx(07)*cx(07)*n07 + cx(08)*cx(08)*n08 + &
         cx(09)*cx(09)*n09 + cx(10)*cx(10)*n10 + &
         cx(11)*cx(11)*n11 + cx(12)*cx(12)*n12 + &
         cx(13)*cx(13)*n13 + cx(14)*cx(14)*n14 + &
         cx(15)*cx(15)*n15 + cx(16)*cx(16)*n16 + &
         cx(17)*cx(17)*n17 + cx(18)*cx(18)*n18 + &
         cx(19)*cx(19)*n19
   Pyy = cy(01)*cy(01)*n01 + cy(02)*cy(02)*n02 + &
                 …
```

Some performance figures

- ✓ Left: time for 256^3 simulation without Smagorinsky Model
- ✓ Right: time for 256^3 simulation with Smagorinsky Model

# Time for section	# Time for section
# init time 0.115159E+01	# init time 0.111767E+01
# loop time 0.716699E+02	# loop time 0.866046E+02
# coll time 0.641728E+02	# coll time 0.791241E+02
# bc time 0.645609E+01	# bc time 0.645086E+01
# diagno time 0.102584E+01	# diagno time 0.101066E+01

- ✓ the Smagorinsky model introduce a 1.23x slowdown!
- ✓ Can it be reduced?

Looking at operation

- ✓ For each Tensor element **`Pij`** we perform
  - 19*3 load operation
  - 19 sums
  - 19*2 products

```
Pxy = cx(01)*cy(01)*n01 + cx(02)*cy(02)*n02 + &
      cx(03)*cy(03)*n03 + cx(04)*cy(04)*n04 + &
      cx(05)*cy(05)*n05 + cx(06)*cy(06)*n06 + &
      cx(07)*cy(07)*n07 + cx(08)*cy(08)*n08 + &
      cx(09)*cy(09)*n09 + cx(10)*cy(10)*n10 + &
      cx(11)*cy(11)*n11 + cx(12)*cy(12)*n12 + &
      cx(13)*cy(13)*n13 + cx(14)*cy(14)*n14 + &
      cx(15)*cy(15)*n15 + cx(16)*cy(16)*n16 + &
      cx(17)*cy(17)*n17 + cx(18)*cy(18)*n18 + &
      cx(19)*cy(19)*n19
```

- ✓ But **`cx(j),cy(j),cz(j)`** can assume only values among **`-1,0,+1`**
  - only 10 non zero values for **`cx(j),cy(j),cz(j)`**

Second optimization:

$$\Pi_{ij}^{(neq)} = \sum_q c_{qi} c_{qj} \, f_q^{(neq)}$$

A compiler can't do that!

```
Pxx =  n01 +n02 +n03 +n04 +n05 +n10 +n11 +n12 +n13 +n14
Pyy =  n01 +n03 +n07 +n08 +n09 +n10 +n12 +n16 +n17 +n18
Pzz =  n02 +n04 +n06 +n07 +n09 +n11 +n13 +n15 +n16 +n18
!
Pxz = -n02 +n04 +n11 -n13
Pxy = -n01 +n03 +n10 -n12
Pyz = +n07 -n09 +n16 -n18
…
```

Drastic operation reduction!

- ✓ **Only 36 sums**
- ✓ No multiplication at all
- ✓ no load of `cx(pop),cy(pop),cz(pop)` vectors

other performance figures

- ✓ Left: time for 256^3 simulation with Smagorinsky Model (no optimized)
- ✓ Right: time for 256^3 simulation with Smagorinsky Model (optimized)

# Time for section	# Time for section
# init time 0.111767E+01	# init time 0.112207E+01
# loop time 0.866046E+02	# loop time 0.730055E+02
# coll time 0.791241E+02	# coll time 0.655008E+02
# bc time 0.645086E+01	# bc time 0.646228E+01
# diagno time 0.101066E+01	# diagno time 0.102758E+01

- ✓ Now the Smagorinsky impact is negligible…

Comment (personal)

- ✓ Compilers can help you, but you must be aware of what they can or cannot do
- ✓ Coding style can help/confuse compilers
- ✓ Optimization capability of today compilers is decreasing
  - more optimization work is demanded to developers
- ✓ Play with different compilers (if possible)
- ✓ Play with different optimization level/flag
- ✓ Use flags to understand what's going on under the hood
  - e.g. `-Minfo` for nvidia compilers

Finite precision

A computer uses finite precision. This means that some rules are different respect infinite precision (standard algebra)

- ✓ **Sum is no more additive**. You can choose three number a,b,c so that

$$\boxed{(a+b)+c \neq a+(b+c)}$$

- ✓ **in finite precision we an “infinite numbers” that behaves like a zero**

$$\boxed{(a+c) = a \to c \equiv 0}$$

Cancellation issue

- ✓ Cancellation happens we we have big and little number to add/subtract
- ✓ Programmer has to take care (with parentheses) of the right order
- ✓ **Compiler can alter the order and produce wrong results**
- ✓ for a = 0.333333 we perform different operation with different values of b

	b = 1.60	b = 6553.6	b = 3355443	b = 1.3421773E+07
b - b + a	0.3333333	0.3333333	0.3333333	0.3333333
b + a - b	0.3333334	0.3334961	0.2500000	0.0000000E+00
a + b - b	0.3333334	0.3334961	0.2500000	0.0000000E+00
a - b + b	0.3333334	0.3334961	0.2500000	0.0000000E+00

- ✓ "Group" quantities with the same range

Floating point: IEEE p754

The **IEEE Standard for Floating-Point Arithmetic** (**IEEE 754**) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units use the IEEE 754 standard.

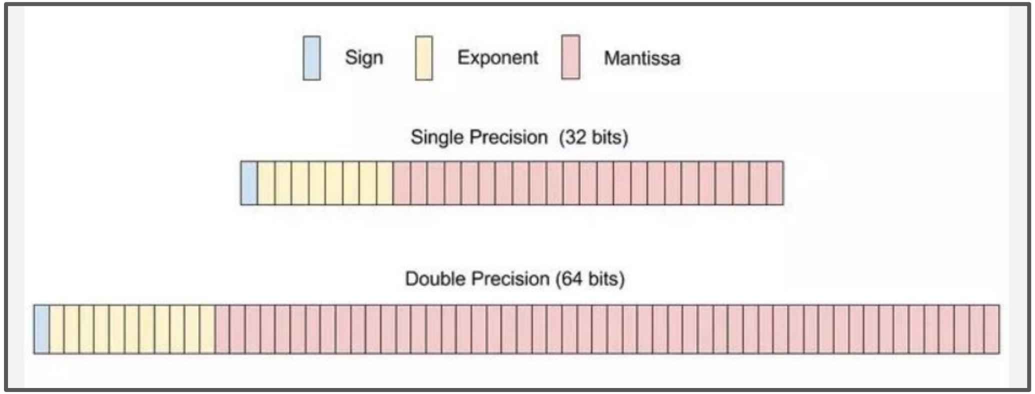
- ✓ ***arithmetic formats****:* sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- ✓ ***interchange formats****:* encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form
- ✓ ***rounding rules****:* properties to be satisfied when rounding numbers during arithmetic and conversions
- ✓ ***operations****:* arithmetic and other operations (such as trigonometric functions) on arithmetic formats
- ✓ ***exception handling:*** indications of exceptional conditions (such as division by zero, overflow, *etc.*)

Floating point: binary representation

✓ Machine Epsilon: minimum non zero value with respect to 1.0

✓ Range: max/min values that can be represented

Precisione	Precisione p	Machine epsilon	Range
Singola	24	$2^{-23} \simeq 1.2 \times 10^{-7}$	$10^{-38} < x < 10^{38}$
Doppia	53	$2^{-52} \simeq 1.2 \times 10^{-16}$	$10^{-308} < x < 10^{308}$



Single vs. Double precision

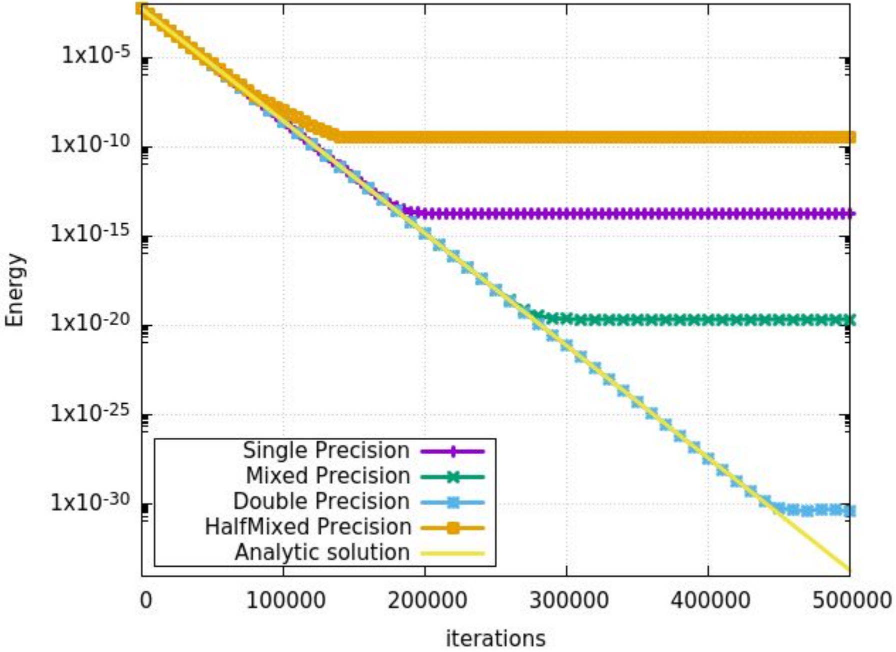
- ✓ Double precision has a wider range respect single precision and is less subject to cancellation issues
- ✓ But today **HW is two times faster** for single precision computation respect double one (do you remember vectorization?)
  - Both CPU and GPU
- ✓ **Half precision is 4 times faster** respect double precision
  - GPU and next generation CPU (it is used for ML)

Hint:

- ✓ Check if single precision is fine for you
- ✓ Use mixed precision: i.e., single precision for stored data, double for all computation
- ✓ Use compiler flags to check if you code is IEEE compliant

TG-Vortex:example

- ✓ Taylor-Green Vortices are a 2D flow configuration where energy decay has an analytical exponential law.
- ✓ Performance (right) in MLUPs, higher is better (GPU)



Precision	MLUPs
Half/Single-	11844
Single	7296
Single/Double	7314
Double/Double	4369

HW trends

- ✓ qqqqqqqqqq
- ✓

Finite precision issues: computing PI

✓ Simple way to compute Pi

$$\frac{\pi^2}{6} = \sum_{n=1}^{\infty} \frac{1}{n^2}$$

```
! forward sum...
     do i=1,elements
        sum1=sum1+(1.0/(float(i)*float(i)))
     end do
!
! backward sum
     do i=elements, 1, -1
        sum2=sum2+(1.0/(float(i)*float(i)))
     end do
…
    write(6,*) 'forward =', sqrt(6.0*sum1)
     write(6,*) 'backward=', sqrt(6.0*sum2)
```

Finite precision issues: computing PI

- ✓ Backward is more precise with respect to forward….
- ✓ Why?

Elements	Backword	Forward
100	0.9969709148	0.9969709907
1’000	0.9996961603	0.9996962361
10’000	**0.9999695955**	**0.9999365829**
100’000	**0.9999969162**	**0.9999365829**
1’000’000	**0.9999997242**	**0.9999365829**
10’000’000	**0.9999999519**	**0.9999365829**

I/O issues

- ✓ I/O operations are really expensive
  - Cycles are order of nanoseconds
  - Memory access are order of 100 or 1000 cycles
  - I/O access can be order of 100000 cycles (milliseconds)
- ✓ Writing on rotating disks implies great latencies

Golden rules

#1: write/read on the disk only if it is mandatory

#2: write/read only what is really important

#3: always use binary and never formatted data (e.g. ASCII) for “huge” dump

I/O Example

- ✓ I/O operations are really expensive

```
! formatted
do k=1,nd
   do j=1,nd
      do i=1,nd
         write(69,*) a(i,j,k)
      enddo
   enddo
enddo
```

```
! unformatted (1)
do k=1,nd
   do j=1,nd
      do i=1,nd
         write(79) a(i,j,k)
      enddo
   enddo
enddo
```

```
! unformatted (2)
write(82)
(((a(i,j,k),i=1,nd),j=1,nd),k=1,nd)
```

```
! unformatted (3)
write(83) a
```

I/O Example

- ✓ I/O operations are really expensive
- ✓ time writing on /scratch
- ✓ Time presented here only as relative reference
  - It heavily depends from the filesystem configuration

	time (sec.)
Formatted	6.88
unformatted-1	1.41
unformatted-2	0.34
unformatted-3	0.10

I/O Some strategies

- ✓ I/O is the bottleneck: avoid it when possible
- ✓ I/O subsystem work with locks: simplify application
- ✓ I/O has its own parallelism: use MPI-I/O or other libraries
- ✓ I/O is slow: compress (to reduce) output data
- ✓ Raw data could be not portable: use library
- ✓ I/O C/Fortran APIs are synchronous: use dedicated I/O tasks
- ✓ Application DATA are too large: analyze it "on the fly", (e.g. re-compute vs. write)

Carefully plan your data production/workflow BEFORE the simulation

Moving Data

Bits per Second Requirements

	1H	8H	24H	7Days	30Days
10PB	25,020.0 Gbps	3,127.5 Gbps	1,042.5 Gbps	148.9 Gbps	34.7 Gbps
1PB	2,502.0 Gbps	312.7 Gbps	104.2 Gbps	14.9 Gbps	3.5 Gbps
100TB	244.3 Gbps	30.5 Gbps	10.2 Gbps	1.5 Gbps	339.4 Mbps
10TB	24.4 Gbps	3.1 Gbps	1.0 Gbps	145.4 Mbps	33.9 Mbps
1TB	2.4 Gbps	305.4 Mbps	101.8 Mbps	14.5 Mbps	3.4 Mbps
100GB	238.6 Mbps	29.8 Mbps	9.9 Mbps	1.4 Mbps	331.4 Kbps
10GB	23.9 Mbps	3.0 Mbps	994.2 Kbps	142.0 Kbps	33.1 Kbps
1GB	2.4 Mbps	298.3 Kbps	99.4 Kbps	14.2 Kbps	3.3 Kbps
100MB	233.0 Kbps	29.1 Kbps	9.7 Kbps	1.4 Kbps	0.3 Kbps

Recap

- ✓ i/O is very slow
- ✓ I/O is very very slow
- ✓ I/O is very very very slow
- ✓ Have I pointed out that I/O is slow already?
- ✓ the compiler is your best friend, but sometimes your worst enemy
- ✓ take care of floating point operation
- ✓ Code refactoring is crucial: a “clean” code is important, but could be inefficient

References

- ✓ D. Goldberg. ‘What every computer scientist should know about floating-point arithmetic’. Computing Survey of ACM, March 1991.
- ✓ M. L Overton. ‘Numerical computations with IEEE Floating Point Arithmetic’. SIAM, 2001.
- ✓ M. Abramowitz and I. Stegun. ‘Handbook of Mathematical Functions’ 1974
- ✓ J.H. Wilkinson. ‘Rounding errors in algebraic processes’. Dover.
- ✓ Dongarra et al. ‘Investigating half precision arithmetic to accelerate dense linear system solvers’