GENERAL FORTRAN OPTIMIZATIONS GUIDE

A training course
of Fortran High Performance Computing
for scientists and developpers

Ryad EL KHATIB
CNRM/GMAP

09.05.2016
TOULOUSE

The purpose of this course is to let you be aware of the performance traps when you code a piece of scientific software in Fortran

Hopefully after the training course + exercise you should be able to write fairly-well-performing code at once !

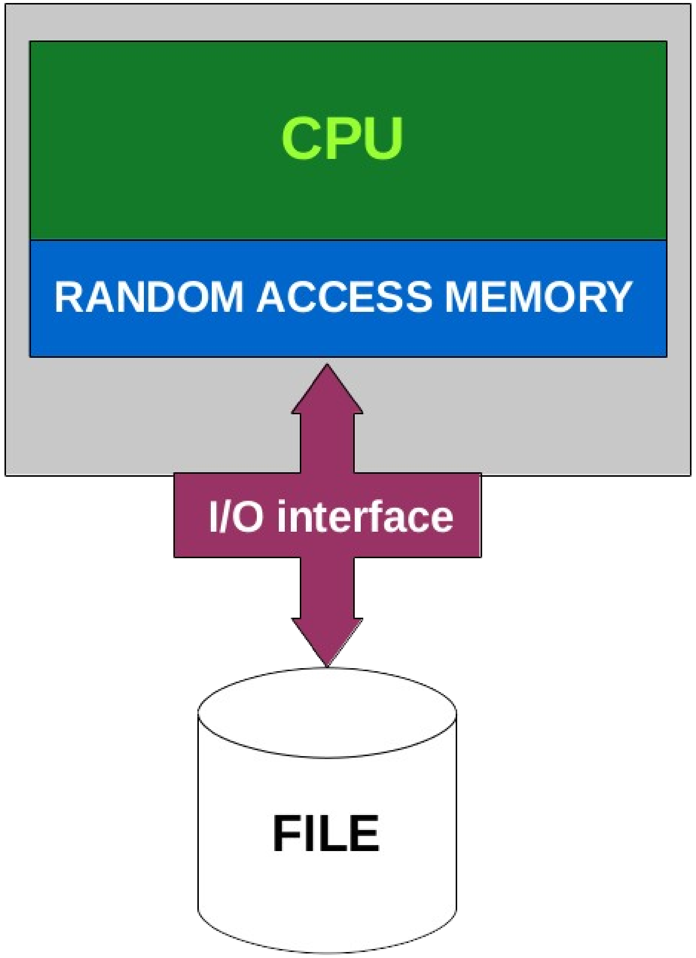
Planning

1. Reminders about High Performance Computers

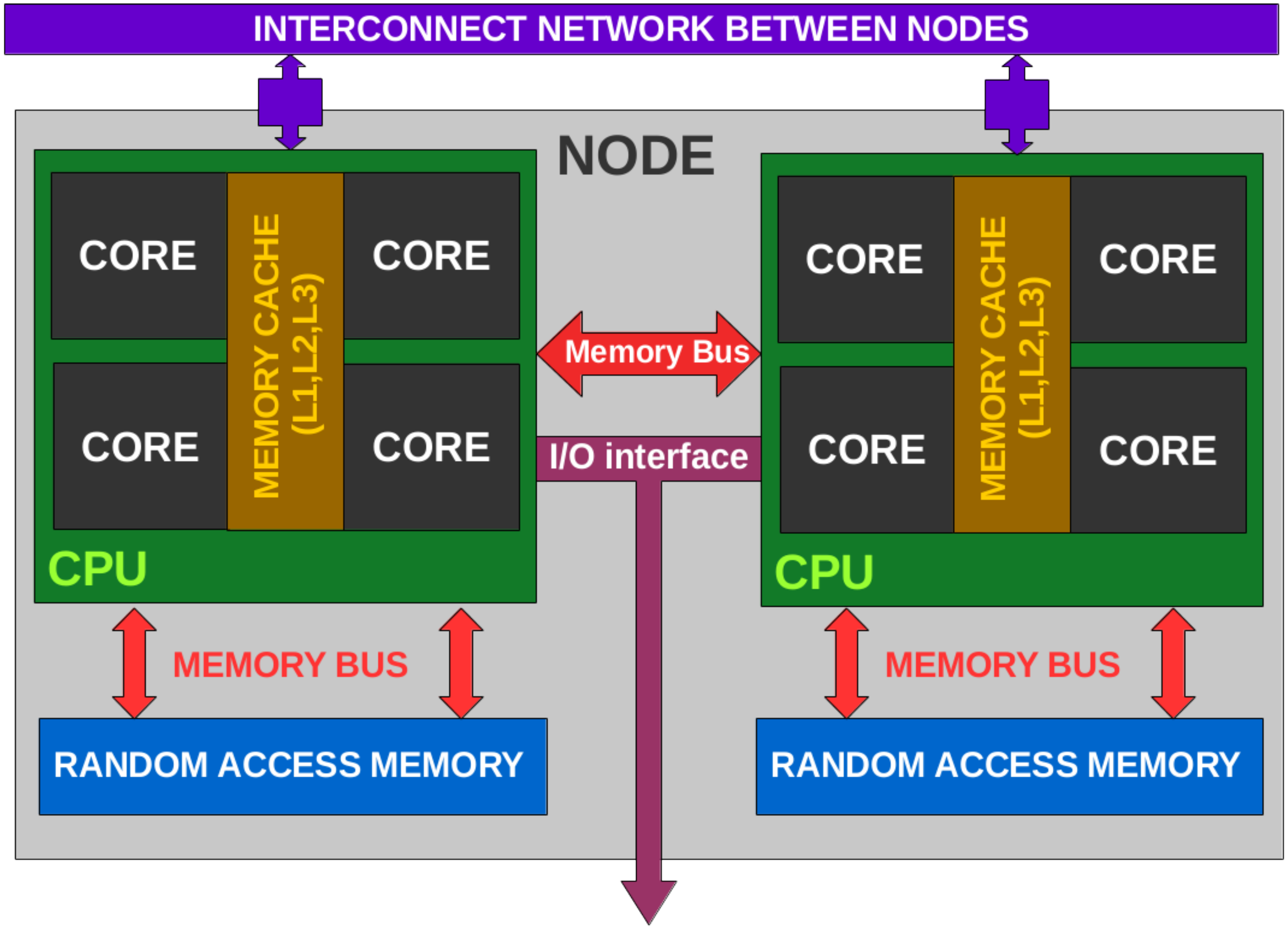
2. Optimization techniques
    - memory caching
    - memory bandwidth saving
    - vectorization
    - memory allocation
    - exercise : *optimize_it !*

3. Profiling Arpege/IFS/Arome

What developers must stop thinking of the (super)computers they are programming on :



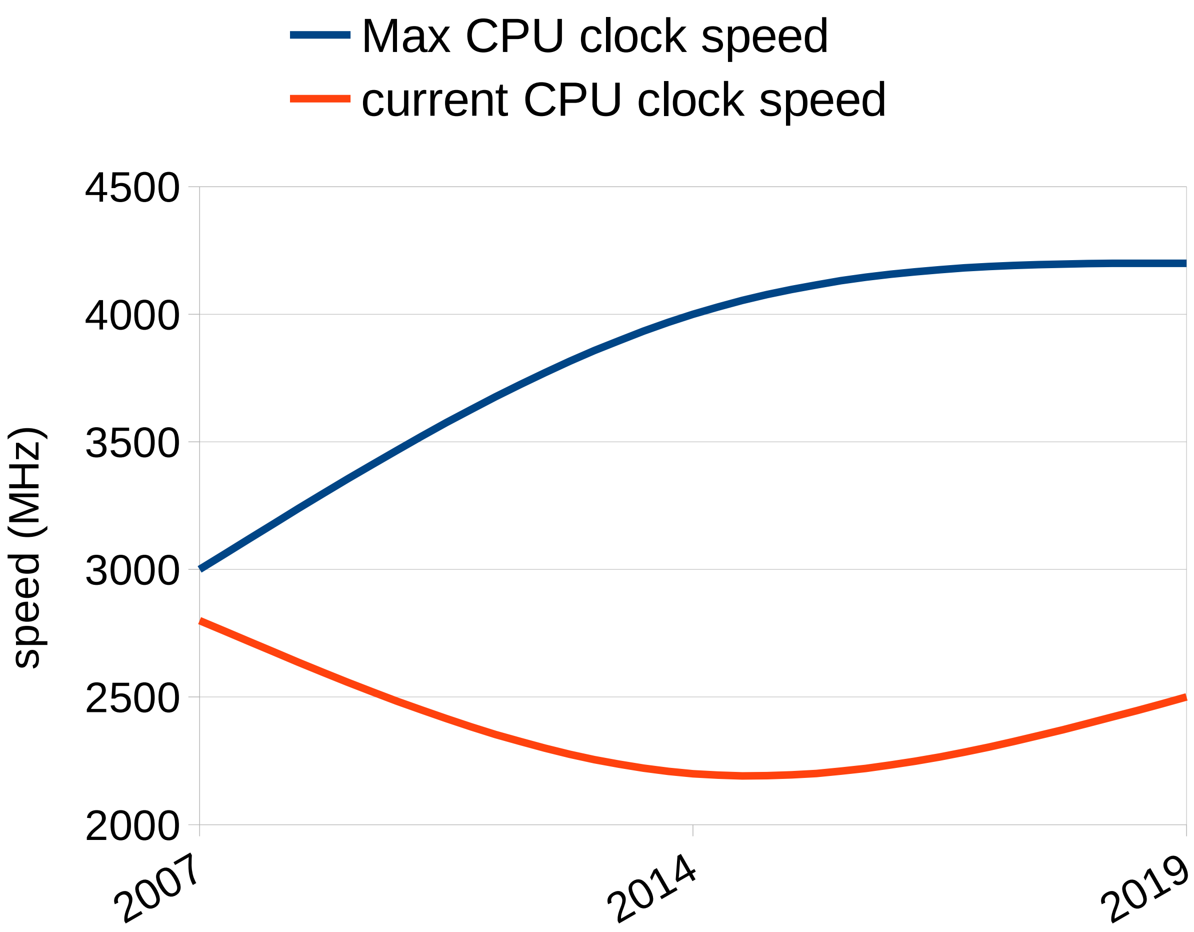
Reminders on High Performance Computers : crude design of a computer node



- **Nodes are interconnected**
- **Each node contains 1 or more CPUs**
- **Each CPU has multiple cores**
- **CPUs access memory via memory bus**
- **All cores of a CPU share a (fast) memory cache**
- **Certain nodes have a direct I/O interface**

Reminders on High performance Computers : evolution of CPUs

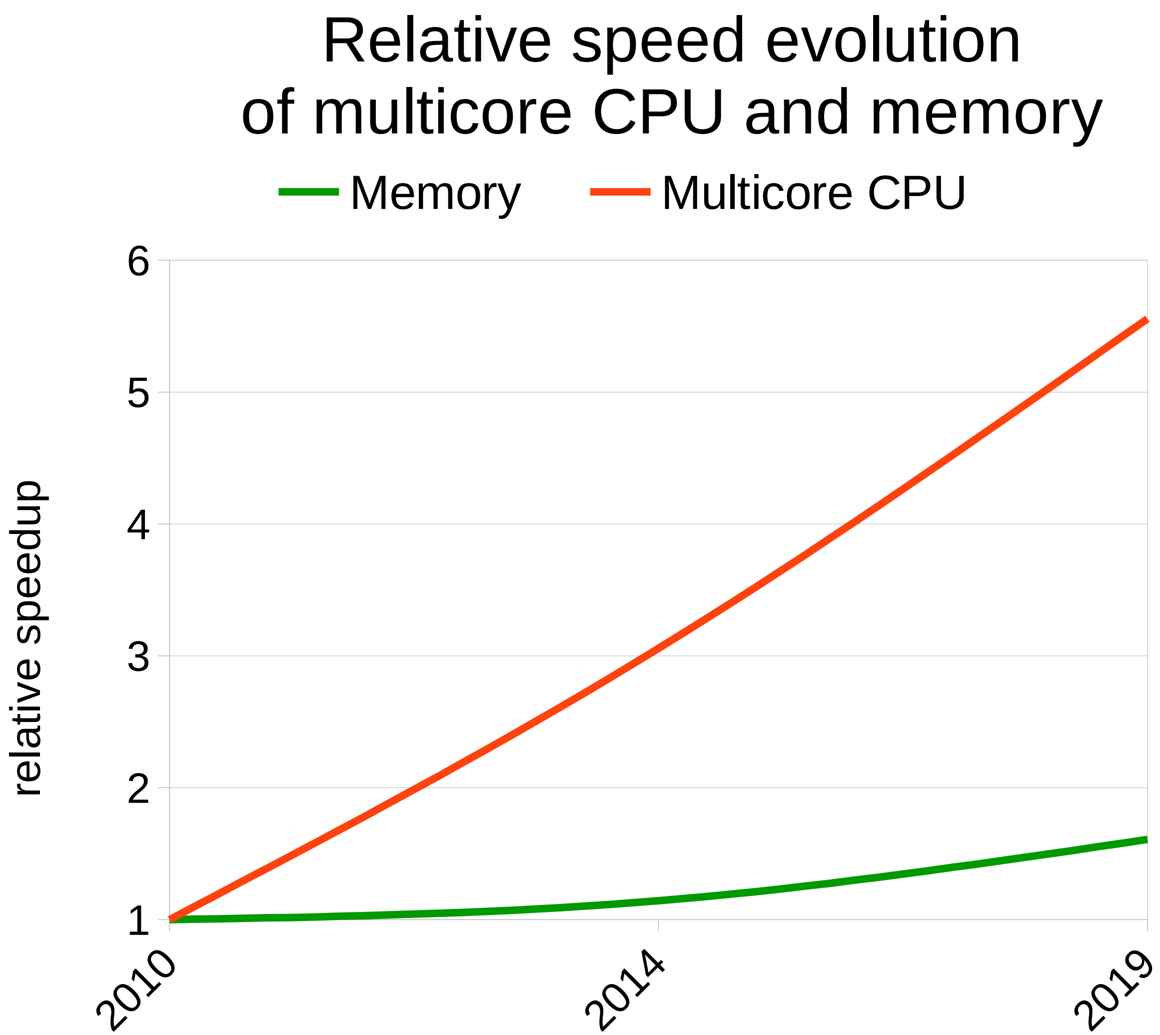
CPU clock speed evolution in past years



- Moore's law is over,
- The improvement of a single CPU core performance is decreasing
- In exchange, the number of cores per CPU is increasing

- => performant programation for a single core is necessary
- => parallel computation is necessary

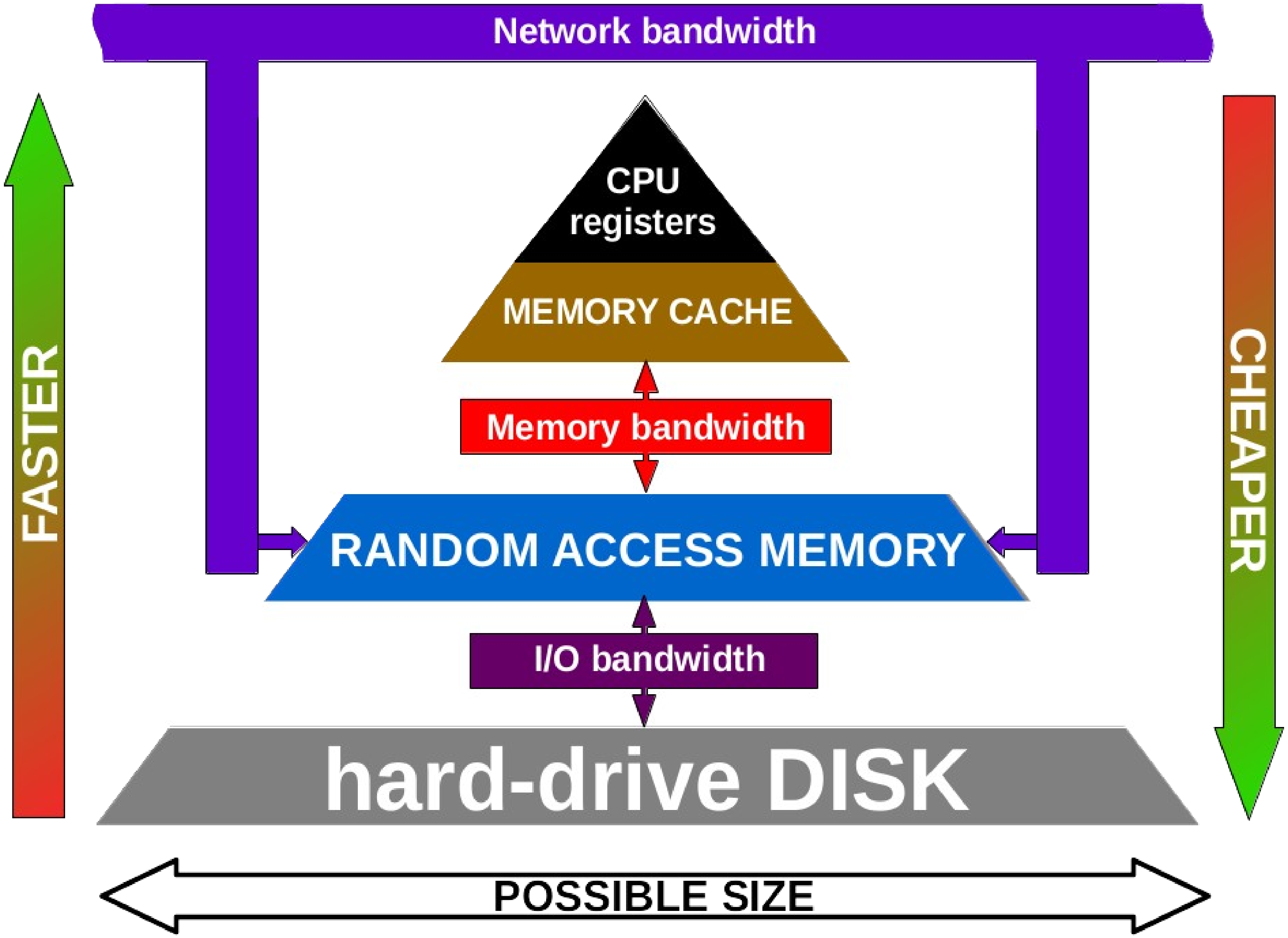
Reminders on High performance Computers : performance evolution of CPUs vs Memory



- Performance of memory is even worse than processors
- (Performance of disk is not better, either)

- => Avoid I/O accesses
- => Avoid memory accesses from Random Acces Memory to CPU or vice-versa

Reminders on High performance Computers : Size, speed (and price) of data storage



"almost all programming can be viewed as an exercise in caching"

Terje Mathisen

… But how to keep my data in the memory cache ???

Cache management policy speculates on locality properties observed in programmation :

- Spacial locality of data :
  if a data is accessed, another data nearby in memory is likely to be accessed at the next instruction
  => fetch it in the cache,
  if necessary drop out what is far
- Temporal locality of data :
  if a memory area is accessed, it is likely to be re-accessed at the next instructions
  => keep it in cache if possible, drop out what has not been accessed for a long time

Memory cache management (1) : Spacial locality

```
REAL :: A(N1,N2), B(N1,N2), C(N1,N2)
DO J2=1,N2
  DO J1=1,N1
    A(J1,J2)=B(J1,J2)*C(J1,J2)
  ENDDO
ENDDO
```

```
REAL :: A(N1,N2), B(N1,N2), C(N1,N2)
DO J1=1,N1
  DO J2=1,N2
    A(J1,J2)=B(J1,J2)*C(J1,J2)
  ENDDO
ENDDO
```

Good spacial locality :-)

A(J1+1,J2), B(J1+1,J2), C(J1+1,J2) are respectively next in memory of A(J1,J2), B(J1,J2), C(J1,J2)

Poor spacial locality :-(

A(J1,J2+1), B(J1,J2+1), C(J1,J2+1) are respectively spaced by N1 variables from A(J1,J2), B(J1,J2), C(J1,J2)

X(1,1)	X(2,1)	X(3,1)	…	X(1,2)	X(2,2)	X(3,2)	...

INNER LOOP ON MOST LEFT-HAND SIDE DIGIT !

memory cache management (2) : Spacial locality

```
REAL, INTENT(OUT) :: A(N1,N2)
REAL, INTENT(IN) :: B(N1,N2)
REAL, INTENT(IN) :: C(N1,N2)
DO J2=1,N2
  DO J1=1,N1
    A(J1,J2)=B(J1,J2)*C(J1,J2)
  ENDDO
ENDDO
```

```
REAL, INTENT(OUT) :: A( :, :)
REAL, INTENT(IN) :: B( :, :)
REAL, INTENT(IN) :: C( :, :)
DO J2=1,N2
  DO J1=1,N1
    A(J1,J2)=B(J1,J2)*C(J1,J2)
  ENDDO
ENDDO
```

Good spacial locality :-)
Arrays contains contiguous data
=> data prefetching
from memory to cache is possible

Unknown spacial locality :-(
Arrays may contain
non-contiguous data
=> prefetching would be unsecure

AVOID IMPLICIT SHAPE DECLARATION !

Memory cache management (3) : Temporal locality

```
REAL :: A(N), B(N), C(N), D(N), Z(N)
DO J=1,N
  A(J)=A(J)*Z(J)
  B(J)=B(J)**2.
  C(J)=A(J)+B(J)
  D(J)=A(J)*B(J)
  Z(J)=Z(J)*D(J)*C(J)
ENDDO
```

Good temporal locality :-)

3 instructions before Z(i) is re-used. Therefore Z is likely to remain in the cache

```
REAL :: A(N), B(N), C(N), D(N), Z(N)
A(:)=A(:)*Z(:)
B(:)=B(:)**2.
C(:)=A(:)+B(:)
D(:)=A(:)*B(:)
Z(:)=Z(:)*D(:)*C(:)
```

Poor temporal locality :-(

3*N +(N-1) instructions before Z(i) is re-used. If the loop is large, Z may be dropped off the cache then fetched again

DO NOT USE ARRAY SYNTAX UNLESS VERY SHORT LOOPS

Memory cache management (4)
Benefits of NPROMA slicing

```
REAL :: A(N,M,K), B(N,M,K), C(N,M,K)
REAL :: D(N,M,K), Z(N,M,K)
DO JK=1,K
  DO JM=1,M
    DO J=1,N
      A(J,JM,JK)=A(J,M,K)*Z(J,JM,JK)
      B(J,JM,JK)=B(J,JM,JK)**2.
      C(J,JM,JK)=A(J,M,K)+B(J,JM,JK)
      D(J,JM,JK)=A(J,JM,JK)*B(J,JM,JK)
      .
      .
       Z(J,JM,JK)=Z(J,JM,JK)*D(J,JM,JK)*C(J,JM,JK)
    ENDDO
  ENDDO
ENDDO
```

X(N*K,M) is replaced by X(N,M,K)

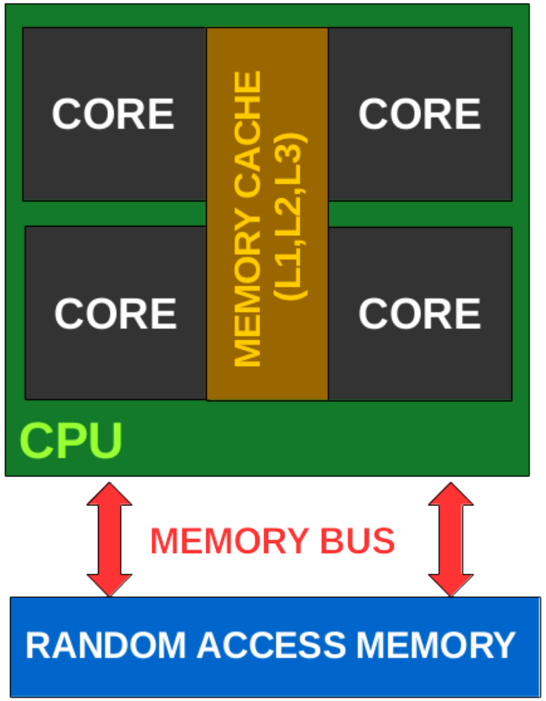
Good for both temporal and spacial locality :-)

Z is likely to remain in the cache whatever the number of instructions in the loop is, provided an adequate value of the leading dimension of the arrays

USE NPROMA AND TUNE ITS VALUE TO FIT THE CACHE SIZE

Memory cache and bandwidth management (1)
Initialization/copies of arrays : problem

```
A(:,:)=B(:,:)
! or
DO JN=1,N
  DO JM=1,JN
    A(JM,JN)=B(JM,JN)
  ENDDO
ENDDO
! or
DO JN=1,N
  A( :,JN)=B( :,JN)
ENDDO
```




```
A(:,:)=0.
! or
DO JN=1,N
  DO JM=1,JN
    A(JM,JN)=0.
  ENDDO
ENDDO
! or
DO JN=1,N
  A( :,JN)=0.
ENDDO
```

Arrays initialisations or copies are rather short loops without data re-use but much data accesses => memory bandwidth and cache are under pressure

How to optimize ?

Memory cache and bandwidth management (2) Initializations/copies of arrays : memory functions

=> elementary, dear Watson : optimize the spacial locality of data !!

A(:)=B(:) is optimized by the compiler
by the use of a specialized function : **memcpy**
memcpy(A,B,size(B)) :
copy the portion of memory used by B over A

A(:)=0 is optimized by the compiler
by the use of a specialized function : **memset**
memset(A,*value*,size(A)) :
initialize the portion of memory used by A with *value*

Of course, the data in arrays should be obviously contiguous !!

Memory cache and bandwidth management (3)
Initialization/copies of arrays : options

```
DO JN=1,N
  DO JM=1,JN
    A(JM,JN)=B(JM,JN)
    Z(JM,JN)=0.
  ENDDO
ENDDO
```

The compiler may not use memset/memcpy :-(

```
A(:,:)=B(:,:)
C( :,:)=D( :,:)
Z( :,:)=0.
```

Optimal use of memset/memcpy to the risk of saturating the cache, causing latencies :-|

```
 DO JN=1,N
   A( :,JN)=B( :,JN)
   Z( :,JN)=0.
 ENDDO
```

Good compromise :-)

Memory bandwidth saving (1)
Recommendation to minimize initializations

```
ZX(:)=0.     <= Useless
ZY(:)=0.     <= Needed

DO J=1,N
  ZX(J)=F(J)
  ZY(J)=ZY(J)+ZX(J)
ENDDO
```

<= OLD CODE

NEW CODE =>

```
INIT0=-1
IF (INIT0 == 0) THEN
  ZVALUE=HUGE(1.)
ELSE
  ZVALUE=0.
ENDIF

IF (INIT0 >= 0) THEN
  ZX(:)=ZVALUE
  ZY(:)=ZVALUE
ENDIF

DO J=1,N
  ZX(J)=F(J)
  ZY(J)=0.
  ZY(J)=ZY(J)+ZX(J)
ENDDO
```

Conditional initialization Allows debugging

INIT0 = 0 : initialization to HUGE
INIT0 = 1 : initialization to a realistic value
INIT0 =-1 : No initialization at all

Necessary initialisation close to calculation => cache re-used

Memory bandwidth saving (2)
Copies of arrays : the help of pointers

```
REAL, INTENT(IN) :: THIS(N)

REAL :: ZTHAT(N)
REAL :: ZX(N)

IF (LALTERNATIVE) THEN
  ZX(:) = THIS(:)
ELSE
  ZX(:) = ZTHAT(:)
ENDIF
```

COPY

```
REAL, INTENT(IN), TARGET :: THIS(N)

REAL, TARGET :: ZTHAT(N)
REAL, POINTER :: ZX(:)

IF (LALTERNATIVE) THEN
  ZX => THIS(:)
ELSE
  ZX => ZTHAT(:)
ENDIF
```

NO COPY

Not always possible, of course … but keep the trick in mind !

Memory bandwidth saving (3)
Copies : the help of pointer remapping

```
SUBROUTINE ARO_MNH(PX)
REAL, INTENT(INOUT) :: PX(KLON,1,KLEV)
END SUBROUTINE ARO_MNH(PX)
```

```
SUBROUTINE DIRECT_MNH(PX)
REAL, INTENT(INOUT) :: PX(:, :, :)
END SUBROUTINE ARO_MNH(PX)
```

```
REAL :: PARO(KPROMA,KLEV)
REAL :: ZMNH(KLON,1,KLEV)

ZMNH(:,1,:)=PARO(:,:)
CALL ARO_MNH(ZMNH)
```

COPY USED

```
REAL :: PARO(KPROMA,KLEV)
REAL :: ZMNH(KLON,1,KLEV)

ZMNH(:,1,:)=PARO(:,:)
CALL DIRECT_MNH(ZMNH)
```

```
REAL :: PARO(KPROMA,KLEV)

CALL ARO_MNH(PARO)
```

NO COPY

```
REAL, TARGET :: &
& PARO(KPROMA,KLEV)
REAL,POINTER :: ZMNH(:, :, :)

ZMNH(1:KPROMA,1:1,1:KLEV) &
& =>PARO(:,:)
CALL DIRECT_MNH(ZMNH)
```

Implicit remapping :
in fortran the address
of the first element is passed

Explicit pointer remapping

Memory bandwidth saving (4)
Arrays swapping : pointers help

```
REAL :: ZARRAY(N), ZBACK(N)

ZBACK(:)=ZARRAY(:)
ZARRAY(:)=F(ZARRAY(:))
ZDIFF(:)=ZARRAY(:)-ZBACK(:)

ZBACK(:)=ZARRAY(:)
ZARRAY(:)=G(ZARRAY(:))
ZDIFF(:)=ZARRAY(:)-ZBACK(:)
```

2 COPIES

```
REAL, POINTER :: ZARRAY(:), ZBACK(:)
REAL, TARGET :: ZYIN(N), ZYANG(N)
LOGICAL :: LLSWAP=.TRUE.
ZBACK(:)=ZARRAY(:)

CALL SWAP
ZARRAY(:)=F(ZBACK(:))
ZDIFF(:)=ZARRAY(:)-ZBACK(:)

CALL SWAP
ZARRAY(:)=G(ZBACK(:))
ZDIFF(:)=ZARRAY(:)-ZBACK(:)
```

CONTAINS

```
IF (LLSWAP) THEN
  ZBACK => ZYIN    ; ZARRAY  => ZYANG
ELSE
  ZBACK => ZYANG ; ZARRAY  => ZYIN
ENDIF
LLSWAP=.NOT.LLSWAP
```

1 COPY

Iterations seen on large arrays in apl_arome ...

Memory cache management : To go further

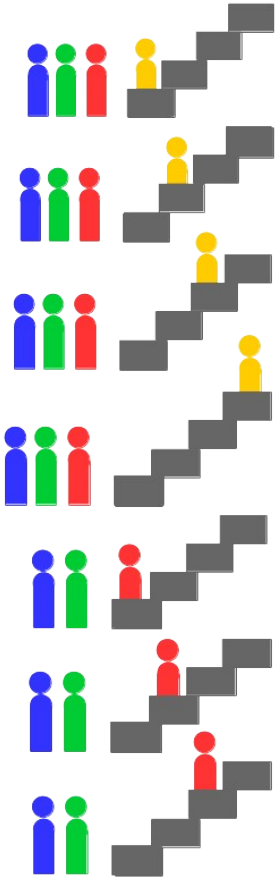
- **There are various algorithms for the couple (hardware, software) to distribute the data in the memory cache :**
  Direct mapping, fully associative, set-associative …

- **There are various algorithms to replace data in the memory cache :**
  Last Recently Used, First In First Out, Random, Last Frequently Used ...

Help the compiler to make the best choice :
Always write the less complex loops you can

Performance enhancement by vectorization

Scalar vs Vector pipelining

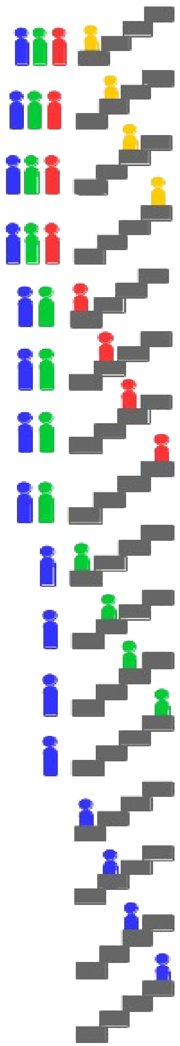


**Analogy of
a vector unit
with mechanic stairs
(against an elevator)**

1 step <=>
1 vector register,
1 instruction,1 cpu clock cycle

The more steps <=>
The more vector registers,
The more potential speedup

Performance enhancement by vectorization



4 dummies, 4 steps : speedup ≈ 2,3

AVX = 128 bits registers
=> 2x 64 bits in double precision
=> maximum speedup = 2

AVX2 = 256 bits registers
=> 4x 64 bits in double precision
=> maximum speedup = 4

So it is worth coding vectorized loops !

Vectorization inhibitors (1) : I/Os

```
REAL :: A(N), B(N), C(N)
DO J=1,N
  A(J)=A(J)*Z(J)
  B(J)=B(J)**2.
  C(J)=A(J)+B(J)
! print for debugging :
  write(nulout,*) ‘test : j=’,j,’C=’,C(j)
ENDDO
```

I/Os break the vectorization !

Don’t put computations and prints in the same loop !

Don’t forget to remove your debugging prints !!!

Vectorization inhibitors (2) : procedures

```
USE MY_MODULE, ONLY : JUNK
REAL :: A(N), B(N), C(N), Z(N)
DO J=1,N
  A(J)=A(J)*Z(J)
  B(J)=JUNK(A(J))
  C(J)=A(J)+B(J)
ENDDO
```

```
REAL :: A(N), B(N), C(N), Z(N)
JUNK(X)=X**3+X**2
DO J=1,N
  A(J)=A(J)*Z(J)
  B(J)=JUNK(A(J))
  C(J)=A(J)+B(J)
ENDDO
```

**Procedures
or external functions
break the vectorization !**

**Use internal functions
or in-line the code,
so that the compiler see
if it can vectorize**

Vectorization inhibitors (3) : math functions

Trigonometric
Functions,
Log, Exp, Sqrt, ... may not vectorise,

```
REAL :: A(N), B(N)
DO J=1,N
  A(J)=LOG(B(J))+B(J)
ENDDO
```

it depends of the compiler :

- **gfortran does not vectorize**
- Cray and NEC compilers vectorize
- Intel compiler vectorizes with compiler specific options :
  `-fast-transcendentals -fimf-use-svml`

```
REAL :: A(N), B(N), C(N)
DO J=1,N
  C(J)=LOG(B(J))
ENDDO
DO J=1,N
  A(J)=C(J)+B(J)
ENDDO
```

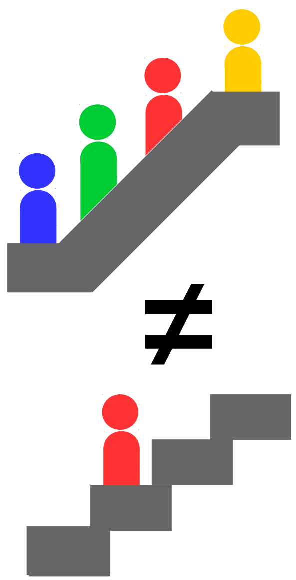
<u>Alternative </u>:
split loop to isolate such functions
(or any external function/procedure)

Vectorization inhibitors (4) : dependencies

```
DO J=2,N-1
  A(J)=A(J-1)+1
  B(J)=B(J+1)*B(J)
ENDDO

DO J=1,N
 A(MASK(J))=A(MASK(J))*B(J)
ENDDO
```

Dependencies, indirect addressing inhibit vectorisation



```
DO J=2,N-1
  A(J)=A(J-1)+1
ENDDO
DO J=2,N-1
  B(J)=B(J+1)*B(J)
ENDDO

!DEC$ VECTOR ALWAYS
!DEC$ IVDEP
DO J=1,N
  A(MASK(J))=A(MASK(J))*B(J)
ENDDO
```

Isolate dependencies,
use compiler directives
if it is worth it, … and correct !

Vectorization inhibitors (5) : conditional pathes

```
DO J=1,N
  IF (A(J) < 0.) THEN
    A(J)=A(J)+1.
  ELSE
    A(J)=A(J)*2
  ENDIF
ENDDO
```

```
DO J=1,N
  A1=A(J)+1.
  A2=A(J)**2
  ZALFA=MAX(0.,SIGN(1.,A(J)))
  A(J)=(1.-ZALFA)*A1+ZALFA*A2
ENDDO
```

Conditional pathes perturbs the vectorization

- **Compilers or developers may vectorize (by masks or compress/expand techniques)**
- **But risk of computation or memory accesses overhead**
- **(… and less readable code)**

Vectorization enhancer : loop fusion

- Loop fusion : reduces loop overhead, by overlapping startup of operations like +, * ; or chaining operations (=> keep data in cpu registers)

FASTER

```
A(:)=B(:)*C(:)
D(:)=E(:)+F(:)
G(:)=A(:)/D(:)
```

Each line is a loop

```
DO J=1, N
  A(J)=B(J)*C(J)
  D(J)=E(J)+F(J)
  G(J)=A(J)/D(J)
ENDDO
```

Overlapping addition and multiplication startup

```
DO J=1, N
  A=B(J)*C(J)
  D=E(J)+F(J)
  G(J)=A/D
ENDDO
```

Memory cache savings

```
DO J=1, N
   G(J)=(B(J)*C(J))/(E(J+F(J))
ENDDO
```

Chaining in CPU registers

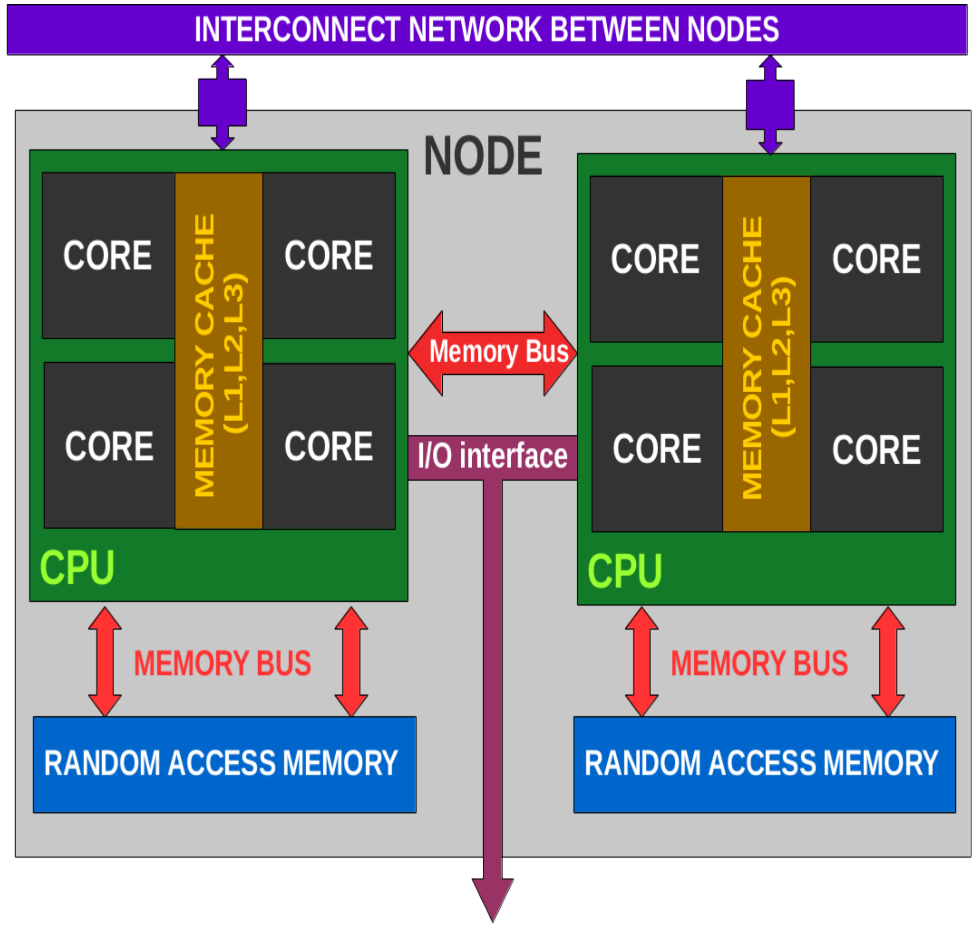
Optimizations Caveats

<u>Remember that :</u>

- At some point memory is shared
- Memory can't go faster than cpu

=> optimization can be desappointing due to memory latencies

- *Don't give up easily*
- *But be pragmatic*

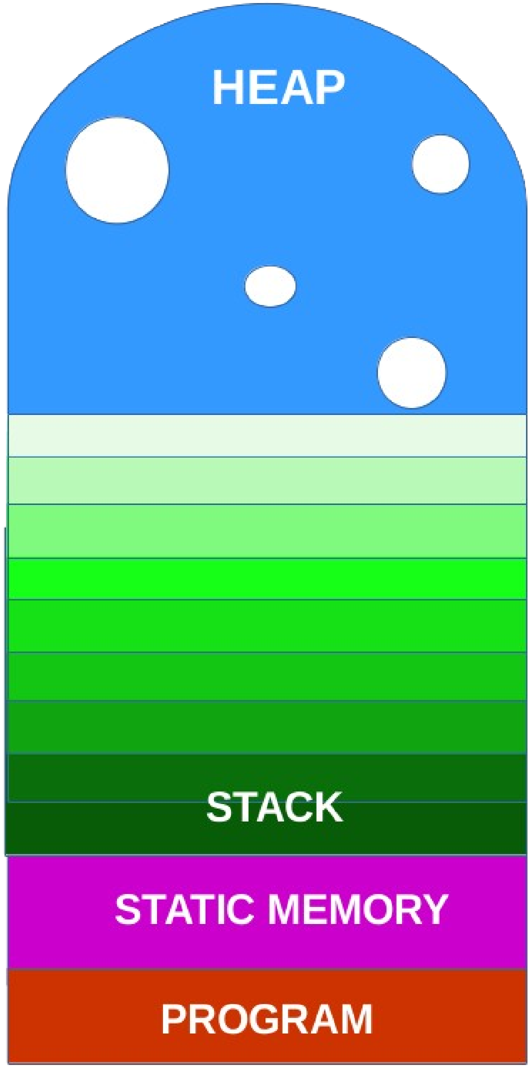


Memory allocations (1) : kinds of allocations

There are 4 kinds of memory allocations :

Static arrays	Allocation at program launch time	Real :: Z(100)	Never deallocated	Very Fast But very memory consuming
Automatic arrays	Allocation at runtime	Real :: Z(N)	Deallocated when the subroutine is left	Fast but memory consuming
Dynamic arrays	Allocation when required	Real, allocatable :: Z(:) Allocate (Z(N))	To be deallocated manually, unless the array is local (F95 std)	Slower But memory-saving
Pointers	Allocation when required	Pointer :: Z(:) Allocate (Z(N))	Must be deallocated manually !!!	Risk of memory leaks and Adressing confusions

Memory allocations (2) : Stack vs Heap



COMPARISON	STACK (automatic arrays)	HEAP (dynamic arrays)
Allocation mode	Last In First Out	Random
Issues	Risk of breaking the stack size limit	Memory fragmentation
Locality aspect	Better	OK
Access time	faster	slower
Programmation	difficult if size is not known in advance	easy
Recommendations	Don't declare huge arrays. Prefer for allocation / deallocation cycles	Allocate once if possible. Avoid allocation / deallocation cycles

Memory allocations (3) : Exemple 1

Cycle of allocation/ deallocation on dynamic array

```
REAL, &
 & ALLOCATABLE :: Z(:)
INTEGER :: N, NITER=6

DO JITER=1,NITER
  CALL FIND_NEWDIM(JITER,N)
  ALLOCATE(Z(N))
  CALL CP(N,Z)
  DEALLOCATE(Z)
ENDDO
```

Removed by a reallocation test (If the size is not supposed to Change all the time) =>

```
REAL, &
 & ALLOCATABLE :: Z(:)
INTEGER :: N, NITER=6

DO JITER=1,NITER
  CALL FIND_NEWDIM(JITER,N)
  IF (ALLOCATED(Z)) THEN
    IF (SIZE(Z) /= N) THEN
      DEALLOCATE(Z)
      ALLOCATE(Z(N))
    ENDIF
  ELSE
    ALLOCATE(Z(N))
  ENDIF
  CALL CP(N,Z)
  ENDDO
DEALLOCATE(Z)
```

Memory allocations (4) : Example 2

Cycle of allocation/ deallocation on dynamic array

```
REAL, &
 & ALLOCATABLE :: Z(:)
INTEGER :: N, NITER=6

DO JITER=1,NITER
  CALL FIND_NEWDIM(N)
  ALLOCATE(Z(N))
  CALL CP(N,Z)
  DEALLOCATE(Z)
ENDDO
```

```
INTEGER :: N, NITER=6

DO JITER=1,NITER
  CALL FIND_NEWDIM(N)
  CALL MYTRICK
 ENDDO

CONTAINS
SUBROUTINE MYTRICK
REAL :: Z(N)
CALL CP(N,Z)
END SUBROUTINE MYTRICK
```

Replaced by an automatic array

Exercise : *optimize_it !*

A small program with 3 parts :

- Initialization of data
- Call to a subroutine written in the ARPEGE style (« F77 loops »)
- Call to a subroutine written in the Meso-NH style (array syntax)

Modify it,
then compile
and execute
with the script
« compile_and_run »

```
 start program
 main =    2.44023514      ini =    3.99390221      arp =    13.4795513      mnh =
8.99044228      checksum =    5.45460847E+12

real	0m29.039s
user	0m26.151s
sys	0m2.802s
```

optimize_it : hints (1)

Use cache-blocking :
with NGPBLKS=4000 and NPROMA=500 the speedup is about 40 %

```
 start program
 main =    2.73156548      ini =    4.04446220      arp =    5.03467751      mnh =
5.54927444      checksum =    5.45460847E+12

real	0m17.430s
user	0m16.771s
sys	0m0.638s
```

optimize_it : hints (2)

In arpegestyle.F90 the loops are not ordered properly, causing memory strides of data. Interchanging the loops speeds up by 8 % more

```
 start program
 main =    2.73328114      ini =    4.06605244      arp =    3.55899620      mnh =
5.55330086      checksum =    5.45460847E+12

real	0m15.962s
user	0m15.162s
sys	0m0.799s
```

optimize_it : hints (3)

In mesonhstyle.F90 isolate the function and make a f77-style loop to overlap operations. Also the various data arrays in a single loop may better spread over the memory cache (… or not, causing cache misses).

Speeds up by 9 % more.

```
  start program
 main =    2.73886108      ini =    4.05294037      arp =    3.55593586      mnh =
4.10793018      checksum =    5.45460847E+12

real	0m14.539s
user	0m13.822s
sys	0m0.684s
```

optimize_it : hints (4)

In inidata the arrays A, D, Y Z don’t need to be initialized.
Speedup : about 10%, though little detrimental effect on arp and mnh (memory cache latency, memory bandwidth insufficient ?)

```
 start program
 main =    2.72255325      ini =    2.29893970      arp =    3.73764133      mnh =
4.16860104      checksum =    5.45460847E+12

real	0m12.972s
user	0m12.441s
sys	0m0.530s
```

optimize_it : hints (5)

In inidata use array syntax on inner loop to force the use of memset. Overall speedup : less than 1% : the compiler may have done the job already by itself. Another compiler may react differently.

```
 start program
 main =    2.71495628      ini =    2.20681286      arp =    3.76890945      mnh =
4.16950417      checksum =    5.45460847E+12

real	0m12.901s
user	0m12.091s
sys	0m0.810s
```

optimize_it : hints (6)

In arpegestyle make a unique vectorized loop by the mean of an inlineable function. Otherwise the function could not vectorize, or the loop would have to be pushed in the function (see mesonhstyle). Speedup : 10 %

```
 start program
 main =    2.72430229      ini =    2.20505905      arp =    2.37683010      mnh =
4.16392517      checksum =    5.45460847E+12

real	0m11.554s
user	0m10.792s
sys	0m0.719s
```

optimize_it : hints (7)

In arpegestyle interface use explicit dimensionning to allow data prefetching in memory cache. Speedup : 5 %

```
 start program
 main =    2.72234344      ini =    2.20254707      arp =    1.78959465      mnh =
4.16979599      checksum =    5.45460847E+12

real	0m10.926s
user	0m10.185s
sys	0m0.740s
```

optimize_it : hints (8)

In mesonhstyle use the same technique of inlineable function to fully vectorize into a single loop, favourising cached data re-use. Speedup : 5 %

```
 start program
 main =    2.71584988      ini =    2.25223923      arp =    1.78060722      mnh =
3.52731037      checksum =    5.45460847E+12

real	0m10.355s
user	0m9.618s
sys	0m0.704s
```

optimize_it : hints (9)

In mesonhstyle the arrays ZA and ZB can now be replaced by scalars, thus saving memory cache occupation.
Speedup : 6 %

```
 start program
 main =    2.70844269      ini =    2.19112682      arp =    1.78019714      mnh =
2.91701508      checksum =    5.45460847E+12

real	0m9.669s
user	0m8.868s
sys	0m0.769s
```

optimize_it : hints (10)

In mesonhstyle merge the last statements to chain data in cpu registers.
Speedup : null. Perhaps the loop is too short, or the compiler did the optimization already.

```
 start program
 main =    2.70068836      ini =    2.19823170      arp =    1.78565693      mnh =
2.91714954      checksum =    5.45460847E+12

real	0m9.643s
user	0m9.013s
sys	0m0.630s
```

optimize_it : hints (11)

In mesonhstyle the use of CONTIGUOUS attribute for the arguments, if one knows that the host program sends contiguous data, would favourise cache data prefetching. Flexibility can be preserved by the mean of a cpp macro. Speedup : 17 % !

```
start program
 main =    2.71683455     ini =    2.20568085     arp =    1.79095125     mnh =
1.21225739     checksum =    5.45460847E+12

real	0m7.967s
user	0m7.257s
sys	0m0.710s
```

Additional exercise (exo2)

Select the most efficient style proposed to perform a loop with a conditional test inside :

- ISTYLE=0 Basic loop with conditional test inside
- ISTYLE=1 Loop with binary mask instead of conditional test
- ISTYLE=2 Split loop
- ISTYLE=3 Split loop + pack/expand

Comment on the performance

Compare the performance results of the flavours :

test_revert_sqrt = test_sqrt with revert conditional test

test_abs = cheap computation on both branches

Additional exercise (exo2) : hints

- ISTYLE=0 Basic loop with conditional test inside :
  the best. « ***Let the compiler do the job, then we see*** »

- ISTYLE=1 Loop with binary mask instead of conditional test
  slower because more computation, especially if one of the branch is expensive (sqrt)

- ISTYLE=2 Split loop
  an attempt to remove tests or to vectorize, which doesn't help. Needs more memory accesses, anyway.

- ISTYLE=3 Split loop + pack/expand
  the worst : we spend more time in data movements than actual computation

Profiling Arpege/IFS/Arome (1)

DrHook is an instrumentation tool in the code which can be activated to profile an application :
export DR_HOOK=1
export DR_HOOK_OPT=prof

'XXX' :
label of this area

ZHOOK_HANDLE :
memory address
of this profiled area

0 : start adress of this area

1 : end address of this area

```
SUBROUTINE XXX

USE PARKIND1 , ONLY : JPRB
USE YOMHOOK  , ONLY : LHOOK    ,DR_HOOK

REAL(KIND=JPRB) :: ZHOOK_HANDLE

CALL DR_HOOK('XXX',0,ZHOOK_HANDLE)
! subroutine body
CALL DR_HOOK('XXX',1,ZHOOK_HANDLE)

END SUBROUTINE XXX
```

Profiling Arpege/IFS/Arome (2)

DrHook can be used to oversample a given subroutine :

- Define one memory handler per profiled area
- Define one label per profiled area

- Region XXX will be whole subroutine minus region A minus region B

```
SUBROUTINE XXX
USE PARKIND1 , ONLY : JPRB
USE YOMHOOK  , ONLY : LHOOK   ,DR_HOOK
REAL(KIND=JPRB) :: ZHOOK_HANDLE
REAL(KIND=JPRB) :: ZHOOK_HANDLEA
REAL(KIND=JPRB) :: ZHOOK_HANDLEB

CALL DR_HOOK('XXX:',0,ZHOOK_HANDLE)

CALL DR_HOOK('XXX:A',0,ZHOOK_HANDLEA)
! subroutine region A
CALL DR_HOOK('XXX:A',1,ZHOOK_HANDLEA)

CALL DR_HOOK('XXX:B',0,ZHOOK_HANDLEB)
! subroutine region B
CALL DR_HOOK('XXX:B',1,ZHOOK_HANDLEB)

CALL DR_HOOK('XXX',1,ZHOOK_HANDLE)
END SUBROUTINE XXX
```

Profiling Arpege/IFS/Arome (2)

- Raw output : 1 text file per MPI task :
  drhook.prof.[1-n]
- Merge profile with **drhook_merge_walltime_max.pl** :
  cat drhook.prof.* | perl -w drhook_merge_walltime_max.pl

```
Number of MPI-tasks : 1040
Number of OpenMP-threads : 5
Wall-times over all MPI-tasks (secs) : Min=876.040, Max=903.840, Avg=883.622, StDev=5.862
Routines whose total time (i.e. sum) > 1.000 secs will be included in the listing
  Avg-%   Avg.time   Min.time   Max.time   St.dev  Imbal-%   # of calls : Name of the routine
  2.10%     18.539      5.275    104.848   14.265   94.97%      1261520 : SLCOMM:SLCOMM_INT
  9.40%     83.099     71.394     97.608    5.059   26.86%      1431040 : TRLTOG
  4.94%     43.671     20.547     72.626   14.135   71.71%      1354080 : TRLTOM
  3.76%     33.226     15.674     71.051    9.935   77.94%      2610400 : TRGTOL
  6.58%     58.130     52.174     66.165    2.342   21.15%      1431040 : TRMTOL
  4.78%     42.276     36.492     48.978    2.003   25.49%     59120736 : FFT992
  3.86%     34.065     22.747     40.554    2.611   43.91%     53334008 : APL_AROME
  0.82%      7.274      1.110     30.232    4.600   96.33%      1249040 : SLCOMM2A
  1.74%     15.412      7.839     27.505    3.797   71.50%     53334008 : RAIN_ICE_OLD
  1.24%     10.973      5.739     26.412    2.645   78.27%      1249040 : SLCOMM2A:SLCOMM2A_INT
  2.68%     23.715     16.035     25.429    1.216   36.94%    800010120 : LAITRI
  1.51%     13.356      2.312     25.427    5.086   90.91%      1249040 : CPG_DRV
```

Beaufix :/home/gmap/mrpm/khatib/benchmarks/tools/drhook_merge_walltime_max

Profiling Arpege/IFS/Arome (3)

- Look to the DrHook profile with and without your source code modifications :

- If a new subroutine has popped up on the top of the list,
- or if the order of most expensive subroutine has significantly change,

- Then your mods must be responsible for something

- => Do not guess what is going on.
  Re-sample your code with DrHook to know more ;
  or if your compiler is talkative, read its optimization report and search for not-vectorized loop or message about memory accesses issues.

Météo-France

ryad.elkhatib@meteo.fr

www.meteofrance | @meteofrance

Memory bandwidth saving (3)
Accumulations : the help of pointers

```
REAL :: ZSUM(NDIM)
REAL :: ZINC(NDIM)

ZSUM(:)=0.
DO JI=1,N
  CALL COMPUTE(JI,ZINC)
  ZSUM(:)=ZSUM(:)+ZINC(:)
ENDDO
```

INITIALIZATION NEEDED

If large sums arrays or many sums here and there ...

```
REAL, TARGET :: ZSUM(NDIM)
REAL, TARGET :: ZINC(NDIM)
REAL, POINTER :: ZARG(:)

DO JI=1,N
  IF (JI == 1) THEN
    ZARG => ZSUM(:)
  ELSE
    ZARG => ZINC(:)
  ENDIF
  CALL COMPUTE(JI,ZARG)
  IF (JI > 1) ZSUM(:)=ZSUM(:)+ZINC(:)
ENDDO
```

NO INITIALIZATION NEEDED