General Fortran optimizations guide

Ryad El Khatib
METEO-FRANCE - CNRM/GMAP

September 2019

Introduction

This document is the support of a training course of Fortran High Performance Computing for scientists and developers. The purpose of this course is to let developers be aware of the performance traps when they code a piece of scientific software in Fortran.
The expectation is that the developers will be able to write fairly-well-performing code at once, without the need of a computer specialist to recode and optimize the software sometimes later to ease the handover to operations.

Contents

Chapter 1

Reminders about High Performance Computers

1.1 Crude design of a computer node

Developers should stop thinking that the (super)computer they are programming on is an assembly of a processor with some memory, and an interface to a disk to read or write a file. Things are more complicated than that, and that simple vision of a computer would mask important hardware parts that count for the computational performance.
A supercomputer is a collection of computer nodes, which look like what is shown on figure 1.1 :

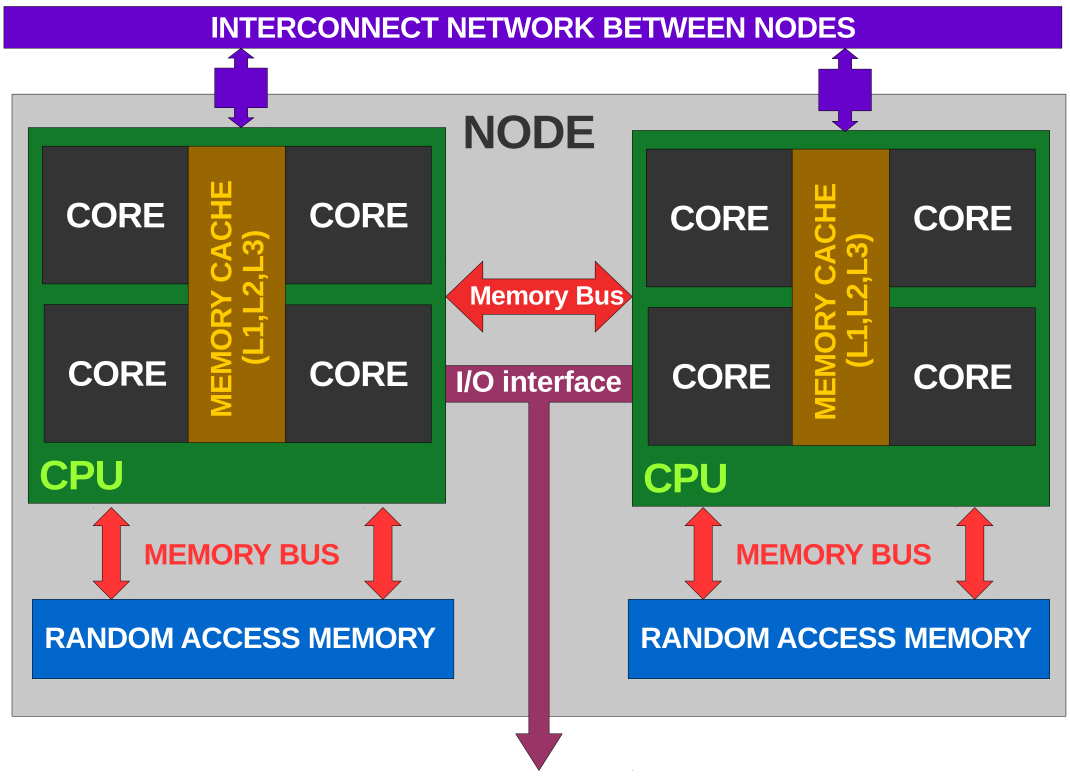

Figure 1.1: *A computer node of a supercomputer*

- the nodes are interconnected with a network with a finite speed for communications

- each node contains one or more processor ("CPU") ; at the moment these lines are written (2019) the usual number of processors per node is two.
- each processor is composed of several "cores". The core is the elementary computational unit. The possible number of cores per processing is growing and growing ; at the moment these lines are written (2019) the order of magnitue is 50.
- processors access the memory via a memory bus which speed is limited (we simply talk about "memory bandwidth")
- all cores of a processor share a piece of fast memory, called "memory cache" (or simply : "cache")
- certain nodes have a direct I/O interface toward a hard drive system. The others will have to transit via an I/O node to access the disks.
- the core of a processor can access the memory of another processor of the same node ; but it will have to transit via another memory bus (that will make the path longer)

1.2 Recent performance evolution processors and memory

The graphic on left side of figure 1.2 shows the recent evolution of processors clock speed : for a single core the speed does not increase any more, an the current speed used is significantly below the maximum value, because higher frequency means more electric energy. In exchange the number of cores per processor is increasing, which is the way of today to increase the computational power without increasing the energy consumption (the problem of electric intensity is replaced by a problem of electric capacity). Therefore the attempts of today are rather toward a reduction of processor clock frequency. In return we have to code efficient and parallel programs.
Considering the processor clock speed is solved by the multiplication of cores, the graphic on right side of figure 1.2 shows now a reasonable speedup. However the problem is not completely solved because the performance of the memory speed (memory bandwidth) is quite bad compared to the clock speed of a multicore processor. This is really an issue because the performance of computation will be limited by the capacity to feed rapidly the processor with data from the memory.

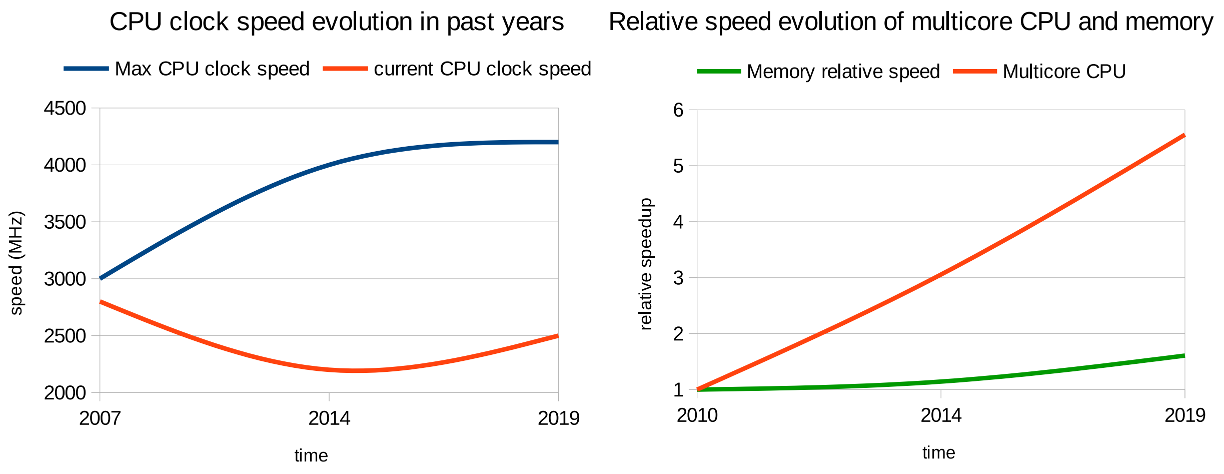

Figure 1.2: *Speed of a single core of CPU (left) and speed of a multicore CPU compared to memory speed (right)*

The performance of disk accesses is not shown on that figure, but it is rather even worse than for memory. Consequently one should limit the frequency of disk accesses, and limit the accesses from the memory to the processors and vice-versa.

1.3 Size, speed (and price) of data storage

Figure 1.3 is based on the classical figure of the pyramid of cost and speed for the various memory supports.

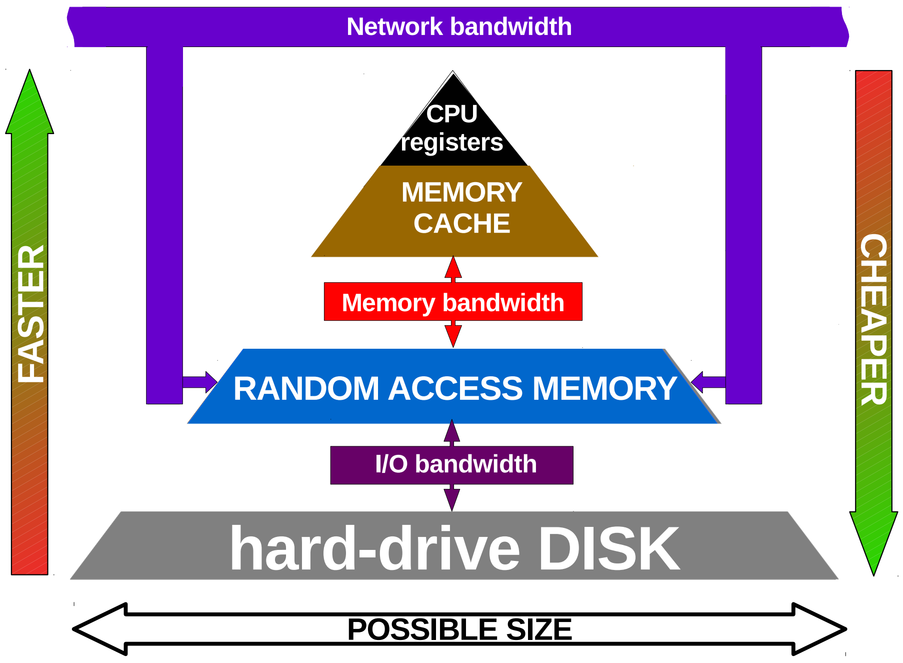

Figure 1.3: *Relative possible size (against speed and cost) of the different memory storage supports*

Disks have the biggest possible size but they are slow and their access is limited by an I/O bandwidth. Random Access Memory is faster, but its access from the CPU is limited by the bandwidth of the memory buses. Also the data in memory exchanged between computer nodes is limited by the bandwidth of the network. Memory cache is a fast, but expensive memory, therefore its quantity is very limited (order of magnitude : Megabytes). Finally CPU registers represents the fastest memory, but it is even smaller.

That figure shows that if we want to compute fast, we should keep the data as much as possible at the top of the pyramid, that is : inside the memory cache. Efficient programming is then an exercice to keep data in memory cache as much as possible.

Chapter 2

Optimization techniques

2.1 Memory caching

The crucial question is then : "... But how to keep my data inside the memory cache ??" While memory is easily understandable by programmers, "cache" is something rather obscure ...
Programmers should be instructed that computers softwares follow a cache management policy ; and this policy speculates on locality properties observed in current programation :

- Spacial locality of data : if a data is accessed, another data nearby in memory is likely to be accessed at the next instruction. As a consequence it can be interesting to fetch nearby data in the cache, thus overlapping memory accesses and computation. And when the cache is filled, far away data may be dropped off.
- Temporal locality of data : if a memory area is accessed, it is likely to be re-accessed at the next instructions. As a consequence it can be interesting to keep that data in the cache as long as possible, and drop off the cache what has not been accessed for a long time.

We shall now see concrete examples of spacial or temporal localities :

2.1.1 Spacial locality

The double loop below is an example of good 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
```

In fortran, data in arrays are stored in such an order that the most left-hand side digit is growing first.
Consequently, `A(J1+1,J2)`, `B(J1+1,J2)`, `C(J1+1,J2)` are stored respectively next in memory to `A(J1,J2)`, `B(J1,J2)`, `C(J1,J2)` and the compiler optimizer, speculating they will be used at the next iteration of the loop, can avantageously prefetch them from memory to the cache.
But if the two loops are interchanged as below :

```
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
```

Then from one iteration of the inner loop to another the variables accessed variables `A(J1,J2+1)`, `B(J1,J2+1)`, `C(J1,J2+1)` are spaced by `N1` variables from `A(J1,J2)`, `B(J1,J2)`, `C(J1,J2)`. Any prefetching of data from these arrays are likely to result in cache misses.
Therefore,

$\boxed{\text{The inner loop must apply to the most left-hand side dimension}}$

Let us now reconsider the same kind of loop where the arrays are dummy arguments :

```
REAL, INTENT(OUT) :: A(N)
REAL, INTENT(IN)  :: B(N)
REAL, INTENT(IN)  :: C(N)
DO J=1,N
  A(J)=B(J)*C(J)
ENDDO
```

That loop has a good spacial locality because the explicit dimensionning of the dummy arrays instructs the compiler that the data in each array are contiguous in memory.
However the Fortran language offer the possibility to write the same code with implicit declarations, which can make the code more robust against bugs :

```
REAL, INTENT(OUT) :: A(:)
REAL, INTENT(IN)  :: B(:)
REAL, INTENT(IN)  :: C(:)
A(:)=B(:)*C(:)
```

Unfortunately that implicit shaped declarations also instruct the compiler that these dummy arrays are not necessarily contiguous in memory. The spacial locality of data being unknown to the compiler optimizer, it may prefer to disable any data prefetching rather to risk cache misses. Consequently that loop will run slower than the former one.
Therefore,

$\boxed{\text{Implicit shaped declarations should not be used}}$

2.1.2 Temporal locality

In the following loop the variable `Z(J)` is re-used after 3 lines of instructions. In order to optimize the cache management by considering the temporal locality of variables, the compiler optimizer is likely to keep it in the cache while other variables from the same iteration `J` may be rapidly dropped :

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

If the same loop is written in array syntax :

```
REAL :: A(N), B(N), C(N), D(N), Z(N)
A(:)=A(:)*Z(:)
B(:)=A(:)**2.
C(:)=C(:)+B(:)
D(:)=D(:)/3.+C(:)
Z(:)=Z(:)*D(:)
```

there are 3*N +(N-1) instruction lines before `Z(i)` is re-used. If the loop is large, `Z` may be dropped off the cache then fetched again. As a result, the loop would run slower, because of an overhead of data accesses.
Note that this is not be a problem if the loop is very short. But knowning that the loop is short may not be enough : the *compiler* should be aware that the loop is short. In other words, the number of iterations of the loop should be a hard-coded variable.
Conclusion :

Do not use array syntax in computational loops

2.1.3 Cache blocking

Cache blocking is a technique to optimize the cache management policy by considering both spacial and temporal locality. Actually this technique is already well known by Arpege/IFS/Arome developers under another name : NPROMA slicing.
The idea is to redistribute the data in the first dimension of an array per chunks : for instance, given a 2D array dimensionned (N*K,M), this array is transformed into a 3D array shaped (N,M,K). At the computation step, an outer loop is added on the third dimension K. Let us now consider the same loop as above, but with this slicing :

```
REAL :: A(N,M,K), B(N,M,K), C(N,M,K), 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)=A(J,JM,JK)**2.
      C(J,JM,JK)=C(J,M,K)+B(J,JM,JK)
      D(J,JM,JK)=D(J,JM,JK)/3.+C(J,JM,JK)
      .
      .
      Z(J,JM,JK)=Z(J,JM,JK)*D(J,JM,JK)
    ENDDO
  ENDDO
ENDDO
```

We can see that for a given chunk JK, `Z` is likely to remain in the cache whatever the number of instructions in the loop is, once an adequate value of the leading dimension of the arrays has been tuned. This leading dimension is what Arpege/IFS/Arome developers know as NPROMA. The best performing value depends on the size of the cache (a computer hardware parameter) and on the program itself.

$\boxed{\text{Use the NPROMA slicing technique and tune its value to fit your program with the cache size}}$

2.2 Memory cache and bandwidth : the case of arrays initializations or copies

Arrays initialisations or copies are rather short loops without data re-use but with much data accesses. Consequently the memory bandwidth and the cache are under pressure.

2.2.1 Recommended style for initializations and copies

No data re-use means that to optimize initializations or copies we should focus on spacial locality of data. For that, one should be aware that there are specialized functions for initialization and for copy : `memset` and `memcpy`. These function are not Fortran intrinsic functions, so we have to write the code in such a way that the compiler will use them (if it is not able to use them by itself). These functins will not consider variables of the code but a segment of memory. For instance :

```
A(:)=B(:)
```

should be interpreted, if these arrays have been declared as contiguous, not as a loop to copy consecutively each element of the array `B` to the array `A`, but as the copy of the memory area covered by `B` to the memory area covered by `A`.
Considering multi-dimensionned arrays, there are three possible style to write the initializations or copies :

1) The pure F77 loop style :

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

With this style, the risk is that the compiler may not be clever enough to substitute the inner loop by a set of calls to `memset` or `memcpy`.

2) The pure array syntax style :

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

This style represents the possibility ot use `memset` or `memcpy` at their best performance ; but there is a potential drawback : the cache may be saturated, causing latencies and hence, detrimental effect on the overall performance.

3) The mixture of the two style :

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

This style appears to be the best compromise, limiting the scope of `memset` or `memcpy` to the first dimension of arrays, which is likely to have been chosen as the optimal value for cache blocking (NPROMA).
This will be the only case where array syntax is (partly) recommended in this document.

Copies or initializations of arrays should be written with array syntax on the inner loop only

2.2.2 Removal of arrays initializations

Still the best way to initialize an array is to analyse first if this array really need to be initialized ; and if not, then remove the initialization. For instance, a dummy argument array with attribute `INTENT(OUT)` should not be initialized ; but one should make sure it will be completely filled while returning from the subroutine. Unfortenately there is a poor habit in the community which consists in initializing any array to zero (or to another "physical" value) to prevent from returning with uninitialized variables or in other words : from returning with incorrect values. But a proper default value should rather be `HUGE`, so that any computation on it would reveal, by a floating point exception (FPE) at runtime, the usage of an uninitialized variable.
In the following (silly) piece of code, the initialization of the array `ZX` is useless, while the initialization of `ZY` is necessary :

```
ZX(:)=0.
ZY(:)=0.
DO J=1,N
  ZX(J)=F(J)
  IF (ZX(J) > 0.) THEN
    ZY(J)=ZY(J)+ZX(J)
  ENDIF
ENDDO
```

The use of a conditional initialization is a safe way to remove the useless initialization of `ZY` while allowing debugging at any time :

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

Then, one should proceed in three steps to validate that code :

1. Set `INIT0 = 1` to reproduce the original code, then switch to `INIT0 = 0`
2. Identify which arrays should be initialized (they cause a Floating Point Exception at runtime) and move them out of the conditional initialization in order to have them unconditionally initialized.
3. Switch to `INIT0 =-1`

The new code will look as such :

```
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.
  IF (ZX(J) > 0.) THEN
    ZY(J)=ZY(J)+ZX(J)
  ENDIF
ENDDO
```

The cherry on the cake is to move the needed initializations of arrays at the very last moment they are needed. In the example above the initialization of `ZY` is moved inside the computation loop, so that `ZY` remains in the cache.

$\boxed{\text{Remove useless initializations of arrays as much as possible}}$

2.2.3 Removal of arrays copies by pointers manipulation

In certain circomstances, pointers can be used instead of arrays copies. A pointer is nothing but an address in memory, therefore pointing to an array is quite much cheaper than coping it. This is particularly interesting when the arrays are big, or if they are copied many times.
This section will describe various examples where copies can be avantageously replaced by pointers manipulations.

Let's start by this simple case :

```
REAL, INTENT(IN) :: THIS(N)
REAL :: ZTHAT(N)
REAL :: ZX(N)
IF (LALTERNATIVE) THEN
  ZX(:) = THIS(:)
ELSE
  ZX(:) = ZTHAT(:)
ENDIF
```

`ZX` can be either `THIS` or `ZTHAT`. Since `ZX` must be "readonly" according to the intent attribute of `THIS`, `ZX` can be easily replaced by a pointer, as follow :

```
REAL, INTENT(IN), TARGET :: THIS(N)
REAL, TARGET :: ZTHAT(N)
REAL, POINTER :: ZX(:)
IF (LALTERNATIVE) THEN
  ZX => THIS(:)
ELSE
  ZX => ZTHAT(:)
ENDIF
```

This is a very simple example of pointer usage ; but it can be applied in more sophisticated circumstances. For instance, it can be used to remove the initialization of a sum :

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

At the first iteration, the sum is actually equal to the increment to add to it ; so the condition to consider will be the iteration number :

```
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
```

Well this case is so simple (the actual computation is confined inside a subroutine) that it can be even re-written without pointers :

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

Here is now a more complex usage of pointers :
In the subroutine apl_arome.F90 there are copies of large arrays in order to accumulate tendencies :

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

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

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

Each time `ZDIFF` is updated, a backup of `ZARRAY` on `ZBACK` is needed. This copy could be removed by interchanging the roles of `ZARRAY` and `ZBACK` at each new step, but that would make the code confusing. Instead, pointers can be used to swap the two arrays, so that `ZARRAY` will be always the result of computation, and `ZBACK` the backup :

```
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(:)
```

The subroutine `SWAP` is contained, and reads :

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

That way, the backup of `ZARRAY` on `ZBACK` is needed only once, to start the swap mechanism.

Consider the use of pointers rather than copying arrays

2.2.4 Removal of arrays reshaping by pointers remapping

Copies of arrays can be seen here and there for reshaping purposes. For instance, subroutine interfaces of the physical parameterizations from Meso-NH model expect 3D arrays (longitudes,latitudes,levels) :

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

while Arpege or Arome handle 2D arrays (NPROMA,levels). Then an array copy for reshaping seems to be necessary :

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

However this copy is just useless because in Fortran (F77 style), to pass an array in argument to a subroutine, just the address of the first element of the array is passed :

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

Somehow the bounds remapping is implicit through the interface. And it also saves a bit of memory !
Well this will not be possible if the interface is using implicit shaped declarations :

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

Then the solution is to use the pointer remapping facility from the Fortran 2003 standard :

```
REAL,  TARGET :: PARO(KPROMA,KLEV)
REAL, POINTER :: ZMNH(:,:,:)
ZMNH(1:KPROMA,1:1,1:KLEV) => PARO(:,:)
CALL DIRECT_MNH(ZMNH)
```

Pointer remapping consists in declaring the lower and upper bounds of each dimension of the pointer, and pointing this definition to a target.

Consider the use of pointer remapping rather than copying arrays

2.2.5 Going further with cache management

Developers should be aware that there are various algorithms for the couple (hardware,software) to distribute the data in the memory cache : direct mapping, fully associative, set-associative ... it is always a compromise between complexity and efficiency.
Developers should also be aware that there are various algorithms to replace data in the memory cache : "Least Recently Used", "First In First Out", Random, "Least Frequently Used", etc.
The role of the developper is then to help the compiler make the best choice. Therefore :

Always write the less complex loops you can

2.3 Vectorization

Vectorization is a technology at the level of the processor which enable to enhance the computational performance of loops.
There are two techniques : "pipelining" and "single instruction multiple data" (SIMD). We are discussing here about vector pipelining.

2.3.1 Description

Vector pipelining against scalar instructions may be compared to mechanic stairs against an elevator (figure 2.1) Graphically, one can see that the mechanic stairs is more efficient to transport a certain number of dummies

Figure 2.1: *Analogy : vector pipelining like mechanic stairs (right) against an elevator for a scalar unit (left)*

because each step can be occupied by a dummy, unlike the (small) elevator which can transport one dummy at a time.
A step of the mechanic stairs can be seen as a vector register, for an instruction and CPU clock cycle. Then the more steps there are, the more speedup is possible. For instance, with 4 steps and 4 dummies to transport like in the figure, 7 vector steps are needed while the scalar unit would require 16 steps. The ratio of these two numbers gives the vector speedup : around 2,3.

Processors using the AVX vector technology have 128 bits registers, which means for programs running in double precision (each real variable is coded on 64 bits) a maximum speedup of 2.
Processors using the AVX2 vector technology have 256 bits registers, which means for programs running in double precision a maximum speedup of 4.

Therefore it is worth coding vectorized loops.

The crucial question is then : "... But how to make my loops vectorized ??" The trivial answer can be "Write simple loops". We shall rather discuss in the next sections what can inhibit the vectorization of a loop.

2.3.2 Vectorization inhibitors : I/Os

I/Os inside loops definitely break the vectorization. The vectorization of a loop can be easily broken by mistake, for instance by forgetting debugging prints :

```
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
```

$\boxed{\text{Don't put computations and I/Os in the same loop. Don't forget to remove your debugging prints !!!}}$

2.3.3 Vectorization inhibitors : loops

If loops are nested, only the inner loop can vectorize.
Therefore when loops are nested the instructions between two loops should not be expensive ; otherwise the loops should be reorganized.
Also if a loop is added inside another loop, only the added loop could vectorized. In the following piece of code :

```
REAL :: A(N1,N2), B(N1,N2), C(N1,N2)
DO J2=1,N2
  DO J1=1,N1
    A(J1,J2)=A(J1,J2)*Z(J1,J2)
    B(J1,J2)=B(J1,J2)*2.
    C(J1,J2)=A(J1,J2)+B(J1,J2)
  ENDDO
ENDDO
```

The inner loop over `N1` is perfectly vectorized. But if a new loop is added inside it, like this :

```
REAL :: A(N1,N2), B(N1,N2), C(N1,N2)
DO J2=1,N2
  DO J1=1,N1
    A(J1,J2)=A(J1,J2)*Z(J1,J2)
    DO J3=1,J2-1
      B(J1,J2)=B(J1,J2)+B(J1,J3)
    ENDDO
    C(J1,J2)=A(J1,J2)+B(J1,J2)
  ENDDO
ENDDO
```

the vectorization of the loop over `N1` is broken in favour of the new loop inside it. That new inner loop may not vectorize, or may vectorize poorly, resulting in an overall loss of performance.

$\boxed{\text{When loops are nested, only the inner loop may vectorize}}$

2.3.4 Vectorization inhibitors : procedures

Procedures (ie: calls to subroutines) or external functions break the vectorization. See the example below :

```
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
```

The compiler has no idea what `JUNK` is ; so it cannot vectorize the loop. Vectorization could perhaps become possible if the compiler can use an interprocedural analysis process, for instance : if the loop and the function are inside the same module.
The vectorization of a loop can easily be broken be a code refactoring, in the idea that a group of source code lines appearing at different places should be transformed into a subroutine.
The solution should be to use internal functions, or inline the code, for instance like this :

```
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(X)
  C(J)=A(J)+B(J)
ENDDO
```

The line `JUNK(X)=X**3+X**2` can be put in a file to include (this is currently done in Arpege), so that it can be re-included in other places :

```
REAL :: A(N), B(N), C(N), Z(N)
#include "junk.func.h"
DO J=1,N
  A(J)=A(J)*Z(J)
  B(J)=JUNK(A(J))
  C(J)=A(J)+B(J)
ENDDO
```

Whatever the solution should be, the point is to give the visibility of the procedure to the compiler, so it can see if the vectorization is possible or not.

Don’t call external subroutines or external functions inside loops

2.3.5 Vectorization inhibitors : math functions

Trigonometric functions, logarithms and exponentials, square root ... may not vectorize, it depends of the compiler. For instance Cray and NEC compilers will vectorize these function. Gfortran would not (at least at the moment these lines are written, considering gfortran 6.1.5). Intel compiler would vectorize if the compiler option `-fast-transcendentals` is specified, but the bitwise reproducibility (ie : obtention of exactly the same results if we change the cache blocking factor `NPROMA` for instance) is not warrantied. To get the bitwise reproducibility the compiler option `-fimf-use-svml` should also be specified, but it is available only since the version 18 of this compiler. Therefore, for High Performance Computation purposes, on should make sure the compiler used has the ability to vectorize these so-called "transcendental math functions". Let us consider the following loop with a compiler not vectorizing such functions :

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

A solution could be simple to split the loop :

```
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
```

But in the particular case, the addition is very cheap compared to the logarithm, so the loop splitting would probably not help. In huge computational loop however, moving the transcendental functions into specific loop would surely help. And with compilers able to vectorize transcendental functions, that kind of loop transformation may not be detrimental.

Transcendental functions may not vectorize with certain compiler

2.3.6 Vectorization inhibitors : Dependencies

Dependencies inhibit the compiler to vectorize ; but this is difficult to figure out. Let us consider the following loop :

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

In our mind we read the execution of this loop sequentially ; so that when `A(J)` is being computed, we think we can use the result of the previous iteration in `A(J-1)`. But this is incorrect in the vector loop because the iteration `J-1` of the loop is not finished, so we cannot speculate that the computed value of `A(J-1)` has been written back to the memory (or the memory cache) : it may still be under computation in the vector registers. Unlike for `A(J-1)`, the value of `B(J+1)` in the second line of the loop can be trusted because at that very moment it is used, it has not yet been modified by the loop so its value is the same as before entering the loop. Then if we split the loop as follows :

```
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
```

the first loop on `A` will not vectorize, while the second loop on `B` will.
There is a way to verify by ourselves if such a loop can vectorize or not : if, when written with array syntax instead of a F77-style loop, it returns the proper result, the it will vectorize. But compilers providing reports on

the code optimization will normally report if a loop has been vectorized or not, and if not, what was the reason, especially dependencies found in loop will be reported. Then the developper, knowing the code in details, may understand if the dependencies are true or if they can be ignored. If we consider the loop below, with indirect addressing via a mask :

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

The compiler has no idea if the mask is creating dependencies or not. If we know that it doesn't (if `MASK` is a bijection, for instance) the we can instruct the compiler to ignore the dependencies in this loop, thanks to a compiler directive (here the directive for the Intel compiler is `IVDEP` for "Ignore Vector DEPendencies") :

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

Check the compiler optimization report for dependencies breaking the vectorization

Vector dependencies can be made ignored by a compiler directive, if the result is safe.

2.3.7 Vectorization inhibitors : conditional pathes

Conditional pathes may not always break the vectorization of a loop, but at least they would perturb it :

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

To vectorize the loop above, the compiler can speculate that it is worth executing each branch along the whole loop and combining the result with a binary mask. This operation can also be coded manually by a developper, like below :

```
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
```

Another technique, for a computationally expensive branche of code, can consist in grouping the data for each branch into arrays, computing each each array, then combining them back. It can also be coded manually by a developer, using the fortran functions `PACK` and `UNPACK`.
The risk in using these techniques explicitely is to cause a detrimental overhead of computation or memory

accesses. Therefore, as a first step, it is better to trust the choice of the compiler. If the loop is particularly complex the compiler may give up on the attempt to vectorize, or the vectorization may be inefficient. Then we can step in, for instance by coding a vectorization by mask, or perhaps on the contrary by using a compiler directive to disable the vectorization.

$\boxed{\text{Loops with conditional blocks may vectorize, or can be vectorized manually if it is worth doing so}}$

2.3.8 Vectorization enhancer : loop fusion

There are also ways to enhance the vectorisation of loops. Especially small loops may be merged ; this is called loops fusion (or loops coalescence). These techniques can reduce the loops startup overhead, the can favourise the overlapping of operations, or chain operations into the CPU registers, reducing the need of the memory cache. Consider the following oiece of code :

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

Each line is a loop ; and a detailed analysis of the first two lines reveals independent operations (multiplication, addition) on different data. When these loops are merged, like below :

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

beside the benefit in memory cache management, the time spent in starting up multiplication and addition, corresponding to independent elecronic circuits, can be overlapped by the compiler optimizer.
Furthermore, the fusion of the former loops makes now the arrays `A` and `D` useless : they can be replaced by two scalar temporary variables, thus saving memory cache space for another usage :

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

Finally, the three lines of the loop can be merge, so that the result of the different operations are chained in the CPU registers, making the code again a bit faster :

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

$\boxed{\text{Fusion of small loops enhances the vectorization}}$

Remark : in case of a huge F77-style loop the reverse manipulation, that is splitting the loop, may improve the performance (this manipulation is called "strip-mining"). However, for a developper it is more difficult to find out if a loop should be split and where to split it, therefore it is better to let the compiler handle the opportunity of strip-mining.

2.4 Memory allocations

Memory can be allocated at different places, depending whether the preference is to limit the total memory amount or to speed up the memory accesses. The kind of allocation depends on that choice.

2.4.1 Kinds of memory allocations

The following array describes the four possible kinds of declarations for an array, and how they are allocated/deallocated :

Kind of array	Declaration	Allocation time	Deallocation time
Static	`Z(100)`	At program launch time	Never
Automatic	`Z(N)`	At runtime	on `RETURN`
Dynamic	`ALLOCATABLE :: Z(:)`	When required	Manual or on `RETURN`
Pointer	`POINTER :: Z(:)`	When required	Manual (deallocate and nullify)

- Static arrays requires that the size is known by parameter (in other words : hard-coded). Their access is very fast but since they are never deallocated, they are very memory-consuming. Better not use them, or use them for small arrays.
- Pointers arrays are handled similarly to allocatable arrays but their usage is heighly discouraged because it can lead to memory leaks or adressing confusion, especially in a parallel environment. One should not forget that a pointer is actually not an array but an address in memory, so that it can be shared by more than one process. Allocated pointers should not only be deallocated but also nullified, otherwise the allocated memory will remain busy. Note that pointers were necessary in Fortran 90 to handle allocatable components of a derived type. Since Fortran 2003 we can use allocatable arrays in derived types.
- Automatic and dynamic arrays are the most flexible supports for allocations. They are allocated on different memory spaces, named respectively "stack" and "heap", which will be described in the next section. Note that since Fortran 95, allocatable arrays which are declared and allocated locally in a subroutine will be automatically deallocated when returning from that subroutine.

2.4.2 Stack versus Heap

Figure 2.2 represent the general memory organization.

- The automatic arrays, allocated on the stack, have the particularity that somehow, arrays are allocated on top of one another ; so that the address for a new array will be quickly found. In return, such arrays will use more memory because it is not possible to deallocate an array in the middle of the stack : all the arrays allocated after will have to be released before, by leaving the subroutines where they have been declared. Excessive usage of the stack (by huge arrays) may eventually break the stack size, which is currently limited on supercomputers which won't allow virtual memory on disk for performance
- The dynamic arrays, on the heap, will be preferably allocated where there are some free space, large enough to host the new array. Allocation/deallocation cycles will provide such opportunity to limit the total amount of memory used, by re-using released memory space. In return the system will spend more time to find the proper free space address.

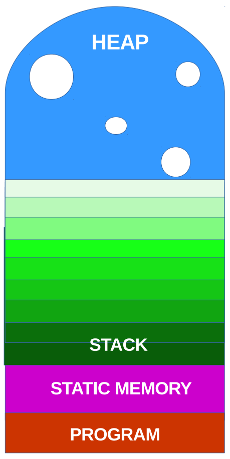

Figure 2.2: *Memory organization*

The following array summerizes the pros and cons of automatic and dynamic arrays :

Comparison	Stack (automatic arrays)	Heap (dynamic arrays)
Allocation mode	Last In First Out	random
Issues	risk of breaking stack size limit	memory fragmentation
Locality aspects	Better	OK
Access time	faster	slower
Programmation	size should be known in advance	easy
Recommendations	Not for huge arrays. Prefer for allocation/deallocation cycles	Suitable for huge array. Allocate only once if possible Avoid allocation/deallocation cycles

2.4.3 Examples of allocation/deallocation cycles

In the following example an allocatable array is reallocated inside a loop, as its size is supposed to change :

```
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
```

To avoid such a allocation/deallocation cycle a first solution is to speculate that the size may not always change. Then the dimension can be tested and the array would be reallocated only if the size as changed :

```
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
```

Alternatively, the array could be reallocated only if the new size is larger than the previous one : it would reduce the number of allocation/deallocation cycles at the risk of using more memory.
Should the array not be huge, then an automatic array can be used. This requires more reorganization of the code, where the computation of the dimension remains in the body but the allocation and main computation is moved to a contained subroutine :

```
INTEGER :: N, NITER=6
DO JITER=1,NITER
  CALL FIND_NEWDIM(JITER,N)
  CALL CP_DRIVER
 ENDDO
CONTAINS
SUBROUTINE CP_DRIVER
REAL :: Z(N)
CALL CP(N,Z)
END SUBROUTINE CP_DRIVER
```

2.5 Before sailing away

Optimization is a difficult job. One can apply the proper recommendations and get desappointing results.
This is because at some point the memory remains shared ; and memory latencies can prevail on the speedup, especially in a multiprocessor environment.
Still, experience tends to prove that when we are ready to spend time to fix a performance issue, we succeed. So, one should not give up easily.
However, the complexity of softwares run on high performance computers are such that we may easily misunderstand the true reason of a performance issue. Also, we may manage to speed up a part of the code, and slow down another one. Well, the point is that the application itself should run faster. Let's be pragmatic !
Last but not least : there are optimizations that will not change the scientific results, and optimizations that can do it. Therefore, after optimizing, make sure the code is correct by extensive validation.

Chapter 3

Profiling Arpege/IFS/Arome

3.1 General recommendations

As explained before, optimization is a difficult job and we can go in the wrong direction easily. Therefore the first thing to do is carefully analyse what is making a program expensive. Profilers are softwares made to analyse a source code performance. Use one to profile your code to identify the bottlenecks, do not guess what is going wrong.
This is an exercise one should do after a code modification : profile before and after the modification. If something has moved in the profiling report, if a subroutine has popped up on the list of the most expensive subroutine, then your modification is likely to be responsible for something.
The compiler should also provide an optimization report for each compiled subroutine. Though it can be difficult to read, it should at least clearly tell if a loop has been vectorized or not, and the optimizations issues it had to cope with.

3.2 Using DrHook as a profiler

"Doctor Hook" is an instrumentation tool used inside Arpege/IFS/Arome, which can be configured as a profiler. It is activated by the following two environment variables :

```
export DR_HOOK=1
export DR_HOOK_OPT=prof
```

Looking down to the source code, we can see that each subroutine invokes `DR_HOOK` twice : at start and at the end of the subroutine :

```
SUBROUTINE XXX

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

REAL(KIND=JPRB) :: ZHOOK_HANDLE

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

```
END SUBROUTINE XXX
```

- The first argument `'XXX'` is the label of the profiled area ; since this area is the whole subroutine the custom is to give to the name of the subroutine to the label
- The last argument is a handler specific to this area, it has to be a real variable of the kind `JPRB`
- the second argument defines the opening (`0`) and the closure (`1`) of the profiling for this area.

DrHook can be used to oversample a subroutine. This is particularly interesting to finely determine which part of a subroutine is expensive. In the following example, the subroutine profile is split in three parts :

```
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

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

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

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

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

Beware that :

- each area should have a different label
- each area should have a different handler

Eventually the profile of the area `XXX` will be the profile of the whole subroutine minus the profiles of the area `A` and `B`.

3.3 DrHook output profile

Each MPI task will produce a text file of its profile ; the default name is `drhook.prof.`*n*, where *n* is the MPI task number (starting from 1). But the custom is to merge all these outputs into one, showing also the load imbalance between the MPI tasks, by the use of a procedure named "`drhook_merge_walltime_max.pl`". The command : `cat drhook.prof.* \| perl -w drhook_merge_walltime_max.pl` should create a text file of this kind :

```
Name of the executable : /scratch/utmp/slurm/khatib.71251544/./MASTERODB
Number of MPI-tasks : 1040
Number of OpenMP-threads : 5
Wall-times over all MPI-tasks (secs) : Min=876.04, Max=903.84, Avg=883.62
Avg-% Avg.time Min.time Max.time St.dev Imbal-% # of calls : Name
2.10%   18.539    5.275  104.848 14.265  94.97%    1261520 : SLCOMM
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
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
1.35%   11.958    6.570   25.045  2.917  73.77%   53334008 : RAIN_ICE
1.22%   10.813    0.405   17.831  4.139  97.73%      14560 : EDIST_SPEC
```

The interpretation in details of the merge profile is complex because the load imbalance has to be considered. But to start with, one should focus on the following points :

- subroutines are sorted by maximum time. This is the best order to cumulate the intrinsic cost of a subroutine and its weight in the load imbalance of the MPI tasks.
- the column "`# of calls`" telling the number of times each subroutine has been called can reveal some computational overhead, caused by the "setup" of the subroutines.
- Focus your attention on the subroutines which do compute, let the subroutines performing communications aside : they come naturally at the top of the profile, not because the communications are expensive but because they spend time waiting for the compute code which is slower on certain tasks. In other words the communications subroutines pay a double penalty for the computational subroutines, which are usually the real responsibles for load imbalance.