Compiling and Debugging Basics

Service CoSiNus IMFT

P. Elyakime
H. Neau
A. Pedrono
A. Stoukov
R. Casta

Outline

• Introduction to the compilation / Modules environment
• Good practices
• Debugging
  ▶ Why?
  ▶ Basic introduction to gdb, valgrind and TotalView
  ▶ Compilation errors and warnings
  ▶ Run time errors and bugs
  ▶ Fortran specificities
  ▶ C/C++ specificities
• Introduction to the optimization
• Validation and sustainability
Introduction to the compilation

• A compiler is a computer program which permits to translate a source code written in a programming language as C/C++/Fortran into binary instructions

• Compilers available at IMFT on linux platforms:
  – GNU (free): gcc, g++, gfortran
  – Intel (2 licenses INPT): icc, icpc, ifort
  – PGI (for fortran only, 2 licenses): pgf77, pgf90
  – Wrappers mpich3 for MPI codes: mpicc, mpicxx, mpif90

Installation

• Gnu compilers : included in linux package (Ubuntu 16.04 LTS, gcc/gfortran version 5.4.0)
• Intel and PGI compilers installed on a centralized server (/PRODCOM), to use it
  
  source /PRODCOM/bin/config.sh  in bash
  source /PRODCOM/bin/config.csh  in csh/tcsh
• Wrappers mpich3 installed on PRODCOM

  FORTRAN:  mympi intel / mympi pgi / mympi gnu
  C/C++:          mympi intel / mympi gnu
Compiling a program

- Options of the compilers available in a shell:
  ```
  man gfortran/gcc/g++
  ifort --help, icc --help, icpc --help
  pgf90 --help
  ```
- Online documentation could also be useful

A simple example:

```cpp
#include <iostream>
int main()
{
    // Post hello and jump a line
    std::cout << "Hello!" << std::endl;
    return 0;
}
```

Different steps of compilation

<table>
<thead>
<tr>
<th>Step 1 : Preprocessing</th>
<th>hello.cpp ➡ hello.i</th>
<th>g++ -E hello.c &gt; hello.i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2 : Compiling</td>
<td>hello.i ➡ hello.s</td>
<td>g++ -S hello.i</td>
</tr>
<tr>
<td>Step 2 : Assembling</td>
<td>hello.s ➡ hello.o</td>
<td>g++ -c hello.s</td>
</tr>
<tr>
<td>Step 4 : Linking</td>
<td>hello.o ➡ a.out</td>
<td>g++ hello.o</td>
</tr>
</tbody>
</table>
• **Step 1 : Preprocessing**
  – “-E” : option of GNU compiler to stop the compilation after the 1st step, the result will be store in the file hello.i
  – The preprocessing make some operations of substitution of the C code:
    • Suppression of the comments
    • Include the files *.h in the files *.cpp
    • Treatment of the compilation directives (#include, #define, #if, «#ifdef, #ifndef, #line, #pragma, #undef, #elif, #else, #endif)
  – The file hello.i is a file text readable !
  – Type `g++ -E hello.cpp > hello.i` to see the result after the 1st step !

• **Step 2: Compiling**
  – “-S” : option of the GNU compiler of the gcc compiler to transform the C, C++ or Fortran code in assembly code (g++ -S hello.i)
  – A file hello.s is automatically product, it’s a readable file but hard to understand if you don’t know the assembly language!
  – “`g++ -S hello.i` to see the file hello.s”!

• **Step 3: Assembling**
  – “-c”: option of the GNU compiler to transform the assembly code in a binary machine code (g++ -c hello.s)
  – Product the object file hello.o unreadable by human. But you can use the command `od -x hello.o` ( post the octet in hexadecimal with the option -x)
  – `od -x hello.o to see the file hello.o in hexadecimal!`
• **Step 4: Linking**
  
  – The file *hello.o* doesn’t include the code of the function «*cout*». This code is in a library and to use it we have to link it in our program.
  
  – **Linking:** *hello.o + Library* ▶️ *executable*
  
  – An executable depends of an operating system: an executable compiled under a Debian Linux doesn’t run under an Ubuntu Linux.

---

### The compilation in practice without library

**Compilation**

- **hello.cpp**

**Link stage**

- **hello.o**

**g++ -c hello.cpp [-o hello.o]**

**g++ hello.o -o hello**

**hello**

---

• **Remark:**
  
  **Compilation + Link stage** in one command:

  ```
  g++ hello.cpp -o hello
  ```

  **hello**
Libraries

- A library is a code including some functions, structures, etc, that we can use in our program
- **Two kinds** of library: **static** (*libxxx.a*) and **dynamic/shared** (*libxxx.so*)
- Use a static library amounts to including the definition of the libraries functions in the executable binary at the link stage of compilation
  - Increase the compilation time and the size of the executable
  - Avoid dependencies
- Use a dynamic library amounts to giving at run time the path where the libraries functions are defined
  - If a library is missing you will have problems
  - The compilation time is faster and the size of executable is lighter

How to link the library to a program?

- A simple example: with the dynamic library OpenCV

```cpp
1 // Declaration of the functions prototypes
2 #include <iostream>    // For the function cout
3 #include <string>      // For the structure string
4 #include "opencv2/core/core.hpp"  // For the structure Mat
5 #include "opencv2/highgui/highgui.hpp"  // For the functions imread and
6                                  // the variable CV_LOAD_IMAGE_UNCHANGED
7
8 int main(int argc, char **argv)
9 {
10   std::string filename_img = argv[1];
11
12   // Print the square root of nbre
13   std::cout << "The images name is " << filename_img << std::endl;
14
15   // Chargement des deux images sous OpenCV
16   cv::Mat in = cv::imread( filename_img, CV_LOAD_IMAGE_UNCHANGED);
17
18   return 0;
19 }
```

- We use the library opencv to read an image!
- **imread, Mat** and **CV_LOAD_IMAGE_UNCHANGED** define in libopencv_core.so and libopencv_highgui.so

- **Linking a library is done in three steps:**
  1. Include at the beginning of our code the file containing the prototype of the function libraries (#include<...>)
  2. Indicate to the compiler where are the prototype function of the library during the compilation step using the option -I
  3. Indicate to the compiler during the linking step where is the code of the library’s functions with the option -L and the name of the library with the option -l

**Compilation with a static or dynamic library**

Add at the compilation step the include path of the library after the option -I

```
g++ -c prog.cpp -I/home/toto/opencv/include
```

Add at the link stage the libpath after the option -L and the libname without the prefix lib and .so/.a after the option -l

```
g++ prog.o -L/home/toto/opencv/lib -lopencv_core -lopencv_highgui -o prog
```

It is possible to compile in one command:

```
g++ prog.cpp -I/home/toto/opencv/include -L/home/toto/opencv/lib -lopencv_core -lopencv_highgui -o prog
```
Difference between static and dynamic library

- With a static library the executable contents the library functions
- With a dynamic library the executable doesn’t contain the definition of the library function, executable needs to know where the libraries are

To indicate the system the location the library paths, we have to complete the system variable **LD_LIBRARY_PATH**

**How to?**
- Fill the LD_LIBRARY_PATH in a shell and export it

  - bash or csh ? **echo $SHELL**
  - Fill the .bashrc/.cshrc:

    ```bash
    export LD_LIBRARY_PATH= PathLib :$LD_LIBRARY_PATH
    setenv LD_LIBRARY_PATH PathLib:$LD_LIBRARY_PATH
    ```

    - Source your bashrc/cshrc: **source ~/.bashrc** or **source ~/.cshrc**

**Comments:**
- The option -l add automatically lib before **prefix** and .so or .a after **prefix**
- `-LPathLib -lopencv_core` can be replace by **PathLib/libopencv_core.so** in the linking step
- The operating system look for libraries automatically in system directories as /lib, /usr/lib, /usr/local/lib, ... and after in the path of LD_LIBRARY_PATH
Another way: use environment Modules

- The environment Modules package provide for the dynamic modification of a user’s environment via modulefiles
- It is a tool that simplify shell initialization and lets users easily modify their environment during the session with modulefiles
- Each modulefile contains the information needed to configure the shell to use an application or to compile or execute your code: it will initialize some environment variables such as PATH or LD_LIBRARY_PATH, ...

Exemple of a modulefile:

```
/PRODCOM/Ubuntu16.04/modules/librairies/mumps/5.0.0/gcc-5.4_mpich-3.2.0:

module-whatis loads the MUMPS library v5.0.0 (gcc 5.4, MPICH 3.2.0)

conflict openmpi
prereq mpich/3.2.0/gcc-5.4

setenv MUMPS_DIR /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0
prepend-path PATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/bin
prepend-path INCLUDE /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/include
prepend-path LD_LIBRARY_PATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/lib
prepend-path LIBRARY_PATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/lib
prepend-path C_INCLUDE_PATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/include
prepend-path FPATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/include
prepend-path CPLUS_INCLUDE_PATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/include
prepend-path LIBRARY_PATH /PRODCOM/Ubuntu16.04/mumps/5.0.0/gcc-5.4_mpich-3.2.0/lib
```
• The Modules package is for a good flexibility: you can load an environment, unload it and load an other dynamically and automatically!

• All popular shells are supported: bash, csh, ksh, sh, ...

• Some commands:
  – `module list`: to see all module ever load
  – `module avail`: to see all module available
  – `module load name_module`: to load a module
  – `module unload name_module`: to unload a module
  – `module purge`: to unload all the modules load
  – `module display`: to show all the variables environment modified by your module

• [https://intranet.imft.fr/Presentation-des-modules](https://intranet.imft.fr/Presentation-des-modules)

How to do to use Module package?

• If you have the line `source /PRODCOM/bin/config.sh` or `source /PRODCOM/bin/config.csh` in your `~/.bashrc` or `~/.cshrc` nothing

• Even if just add this line in your bashrc profile or cshrc profile!
Samples

• To load mumps in your shell

$ module load mpich/3.2.0/gcc-5.4

$ module load mumps/5.0.0/gcc-5.4_mpich-3.2.0
$ ERROR:151: Module 'mumps/5.0.0/gcc-5.4_mpich-3.2.0' depends on one of the module(s) 'mpich/3.2.0/gcc-5.4'
mumps/5.0.0/gcc-5.4_mpich-3.2.0(13):ERROR:102: Tcl
command execution failed: prereq mpich/3.2.0/gcc-5.4

$ module load mpich/3.2.0/gcc-5.4 mumps/5.0.0/gcc-
5.4_mpich-3.2.0

Outline

• Introduction to the compilation /Modules environment
• Good practices
• Debugging
  ▶ Why?
  ▶ Basic introduction to gdb, valgrind and TotalView
  ▶ Compilation errors and warnings
  ▶ Run time errors and bugs
  ▶ Fortran specificities
  ▶ C/C++ specificities
• Introduction to the optimization
• Validation and sustainability
Good practices for your developments

• Avoid too long source files!
• Use a makefile if you have more than one file to compile
• In Fortran: "implicit none" mandatory at the beginning of each program, module and subroutine!
• Use compiler’s check options

Outline

• Introduction to the compilation / Modules environment
• Good practices
• Debugging
  ➔ Why?
  ➔ Basic introduction to gdb, valgrind and TotalView
  ➔ Compilation errors and warning
  ➔ Run time errors and bugs
  ➔ Fortran specificities
  ➔ C/C++ specificities
• Introduction to the optimization
• Validation and sustainability
Why talk about debugging?

Yesterday, my program was running well:
% gfortran myprog.f90
% ./a.out
% vmax= 3.3e-2

And today:
% gfortran myprog.f90
% ./a.out
% Segmentation fault

Yet I have not changed anything...
Because black magic is not the reason most often, debugging could be helpful! (If you really think that the cause of your problem is evil, no need to apply to CoSiNus, we are not God!)

What is debugging?

A methodical process to find and fix bugs in a code

⇒ It is useful to follow this process even if your results seem OK

⇒ Debugging process is a way to improve the robustness of your code

⇒ Improve debugging technics permits to save time. (20% of time spent to code, 80% to debug!)
Outline

- Introduction to the compilation/Modules environment
- Good practices
- Debugging
  - Why?
  - **Basic introduction to gdb, valgrind and TotalView**
  - Compilation errors and warning
  - Run time errors and bugs
  - Fortran specificities
  - C/C++ specificities
- Introduction to the optimization
- Validation and sustainability

GDB

- GDB is the GNU debugger, available with gcc/gfortran but also Intel/PGI
- GDB is able to execute program step by step watching desired variable, break when a condition is true or at specific line
- GDB is able to modify a variable on the fly
- GDB is able to backtrace an error to provide more information on it
- How to use gdb?
Basic use of gdb (1)

% gfortran -c *.f90  
% gfortran buggy_program fill_array.o -o buggy.exe

% buggy.exe 5  
Segmentation fault

% gdb buggy.exe
(gdb) run 5
Program received signal SIGSEV, Segmentation fault.
0x0000000000403f3c in fill_array ()

Basic use of gdb (2)

% gfortran -c -g *.f90  
% gfortran buggy_program fill_array.o -o buggy.exe

% buggy.exe 5  
Segmentation fault (core dumped)

% gdb buggy.exe
(gdb) run 5
Program received signal SIGSEV, Segmentation fault.
0x0000000000403f3c in fill_array
(x=.., i1=1, i2=300, j1=1, j2=error reading variable: Cannot access memory at address)
Valgrind

Valgrind is a powerful memory checking tool. It is able to catch:
- use of uninitialized values
- out of bounds access
- stack overflow
- memory leaks
- ...

Basic use of valgrind

Executable compiled with –g
% valgrind ./a.out
LEAK SUMMARY:
  definitely lost: 40 bytes in 1 blocks
  ...
  still reachable: 32 bytes in 1 blocks
Memory profiling with valgrind

Executable compiled with -g
% valgrind --trace-children=yes --tool=massif ./a.out
% nm_print massif.out.* | less

Command: a.out
Massif arguments: (none)
m_s_print arguments: massif.out.8988

Introduction to TotalView

https://intranet.imft.fr/TotalView
8 tokens, educational licenses
- Compile the code with -g option
- Run TotalView > totalview

Run totalview in foreground mode (avoid totalview &)!
- In « Program » tab, click one « Browse » to chose executable (compiled with -g)
- At IMFT, for a MPI code, in Parallel tab :
  - Parallel System : chose MPICH3
  - Tasks : choose process number (<=8)
Outline

- Introduction to the compilation /Modules environment
- Good practices
- Debugging
  - Why?
  - Basic introduction to gdb, valgrind and TotalView
  - Compilation errors and warning
  - Run time errors and bugs
  - Fortran specificities
  - C/C++ specificities
- Introduction to the optimization
- Validation and sustainability

Errors at compiling step (1)

Read the error or warning message: it gives the file and the line of the error:

- Syntax errors: easy to fix
- Warning messages: understand and fix them!
- Link stage: library issues
- Cross-compiling (code mixing C and Fortran no deals here)

Intel and gnu compilers are verbose.
Pgf90 is a (too) permissive compiler.
Using different compilers provides a more robust code.
Error at compiling step (2)

real (kind=8), dimension(nx, ny) :: psi

% ifort example.f90
example.f90(32): error #6351: The number of subscripts is incorrect.
[PSI]
    psi(1) = 0.0
compilation aborted for example.f90

% pgf90 example.f90
PGF90-W-0155- The number of subscripts is less than the rank of psi
(example.f90: 32)
0 informations, 1 warnings, 0 severes, 0 fatal for stommel

% gfortran example.f90
example.f90: 32.7:
psi(1) = 0.0
Error: Rank mismatch in array reference at (1) (1/2)

Error at compiling step
INCLUDE files missing

% head buggy_program.cpp
#include " buggy_program.h"

% g++ -c buggy_program.cpp
buggy_program.cpp: fatal error: buggy_program.h: file not found

If the included file is not in a standard location
(/usr/include, /usr/local/include, ...), or in the same
directory than the source file, the include file path has to be
specified with -l.
% g++ -c buggy_program.cpp -l /home/logi n/include
Error at link stage: library issue

Example: how to link a code with different lapack libraries?

To link with the compiler’s lapack default library:

- **gfortran**: `main.o sub1.o -llapack -o exe`  
  - **ifort**: `main.o sub1.o -mkl -o exe` # Math Kernel Library  
  - **pgf90**: `main.o sub1.o -L/$(PG)/lib -llapack -o -exe` # PG Math Library

To link with a user defined directory:

- **gfortran**: `main.o sub1.o -L/home/logi/n/ibs -llapack -o exe`  
  
  Equivalent to:

- **gfortran**: `main.o sub1.o /home/logi/n/ibs/liblapack.a -o exe`

Outline

- Introduction to the compilation/Modules environment
- Good practices
- Debugging
  - Why?
  - Basic introduction to gdb, valgrind and TotalView
  - Compilation errors and warning
  - Run time errors and bugs
    - Fortran specificities
    - C/C++ specificities
- Introduction to the optimization
- Validation and sustainability
Runtime bugs

- Floating point exception (FPE)
- Incorrect numbers (NaNs, inf, infinity)
- Segmentation fault (SIGSEV)
- Not enough memory
- Aborted
- Illegal instruction (SIGILL)
- Execution crashes without error message

Compilers options

When a run time bug occurs, the first thing to do is to compile the code with generic debugging options:

- `-g -traceback` (Intel, PGI)
- `-g -fbacktrace` (gfortran)
- `-g` (gcc/g++)

It is easier to correct a code during compilation stage than to debug a run-time error. Don’t hesitate to activate warning at compilation:

- `-Wall` or `-Wextra` (gcc/g++/gfortran)
- `-warn` (intel)
- `-Minform=inform` (PGI)
-g compilation option

The MOST important option to debug!
Whatever the compiler, -g option produces debugging information. It is possible to use it all the time:
Intel & GNU allows to use -g with an optimization option (ex. -O3) but implies -O0 without optimization option
PGI: -g disabling optimization, -gopt without disabling optimization

Floating point exception and incorrect numbers

**Floating point exception, FPE or incorrect numbers generated: Infinity, NaN**

Arithmetic errors :
- divide by zero (uninitialized variables)
- Modulo 0
- arithmetic error value out of bounds :
  \( \text{asin}(3.0), \text{sqr}(1.0), \log(-3.0), ... \)

⇒ Compile with debugging options
⇒ Use gdb at execution to find the line in the code
Segmentation fault (1)

**SIGSEV, Segmentation Fault, Memory Fault**

« A **segmentation fault** or **access violation** is a crash caused by an attempting of access memory nonexistent or protected »

⇒ Compile with debugging options
⇒ In Fortran: compile with check bounds option
⇒ Use gdb at execution to precise the problem
⇒ Use valgrind if necessary

Segmentation fault (2)

Possible causes of a segmentation fault:
- Stack overflow
- Buffer overflow
- Array out of bounds
- Dereference a NULL pointer: use a pointer affected to NULL and change it
- Attempting to access memory the program does not have rights to (the kernel of the OS)
- Attempting to write read-only memory
Stack overflow

Bug caused by a process that attempts to write outside of the space allocated to the call stack.

**Call stack**: area memory to record some information about the active function in a program (locale variable, parameter, ...)

Example:
- Infinity recursive function
- Allocate a too large static variable in the call stack

=> to avoid stack limitation:
  - bash: ulimit –s unlimited
  - csh: unlimit stacksize

=> use dynamic allocation for large arrays when it is possible

Common issue on occigen CINES supercomputer!

Buffer overflow

Bug caused by a process that attempts to write outside the space allocated for the buffer.

**Buffer**: area memory of the RAM used to put temporarily data between two processes not working at the same tempo. Buffers are used to regulate input-output.

gdb will catch it with backtrace

can ask gcc to ignore it using : -fno-stack-protector
Wrong results

Many errors can provide wrong results:
- uninitialized value
- algorithm issues
- precision loss
- ...

Most of these can not be found with debugging, be smart!

Not enough memory

- O: ALLOCATE: 18446744073709551615 bytes requested, not enough memory
  - Code stops without error message
  - BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES EXIT CODE: 9
  - Killed (signal 9)

May due to:
- a program needing a large amount of memory
- a memory leak

⇒use « t op » to check available resources and needs of the code
⇒use valgrind --leak-check=full or memory check of TotalView
Aborted: resource starvation

Abort, supprimé

Quota overflow (no enough space on the disk where I/O are written, / tmp directory full, full swap space)

- Check disk space with « df » command
- Check the swap with « top » and restart the computer to clean up the swap if necessary

Illegal instruction

SIGILL, Illegal instruction
Instruction non permise

Most often this error is due to a code compiled with optimized options incompatible with the target host. Example of target dependent optimization options:
- t p=core2, -Mvect =sse, -sse, -Avx, -fast

=> Recompile after removing all target dependent optimization options!
Fortran’s specificities

• I/O issues
• Compilation by default does not check all possible errors (permits more bugs than C/C++ language)
• Lots of compiler’s options exist and permit to force a compilation error or a run-time crash when a bug occurs.
• Most frequent errors: out of boundary, division by a too small number (~0), consistency of subroutines call and formal arguments
I/O issues

Severe (24): end-of-file during read, unit 9
Severe (67): input statement requires too much data, unit 9

Error messages given by compiler at execution are often explicit enough to solve the problem.
=> check the number of data to read
=> check the data type (real 4, real8, ...)
=> be careful with little/big endian (big endian is less used now)

Check floating point exception

intel: -fpe0
pgf: -Ktrap=f p
gnu: -ffpe-trap=invalid, zero, overflow, underflow, precision

Stop the execution when a floating point exception error occurs.
(Needs to use gdb to find the line)
% pgf 90 -g -Ktrap=f p example.f90
% gdb a.out
(gdb) run 1
Program received signal SIGFPE, Arithmetic exception.
0x000000000000000d in do_jacobi (psi = ..., new_psi = ...,
diff =26820002, i 1=1, i 2=300, j 1=1, j 2=300, ierr=1)
   at ./example.f90: 57
57   psi ( 1, 1) =1.0/ psi ( 0, 0)
Check bounds

intel:  - CB
pgf:  - C
gnu:  - f bounds- check

%ifort -g example.f90
.%./a.out
forrtl: severe (174): SIGSEGV, segmentation fault occurred

%fort -g -CB example.f90
.%./a.out
forrtl: severe (408): fort: (2): **Subscript #2 of the array PSI has value 310 which is greater than the upper bound of 301**

Check uninitialized variables

intel:  - ftrapuv
pgi:  no option available => use valgrind
gnu:  - finit-real =snan -ff pe-trap
(or  -Wuninitialized)

Uninitialized variables could provide different run-time errors: Segmentation Fault, Floating Point Exception or wrong results.
Wrong results

A wrong result could have many causes. In Fortran you can:

• Check data type error
• Check the number, dimension and kind of a each argument in routine calls (it is possible to use interfaces in a fortran module)
• Use a debugger...

Fortran debugging options: a different cocktail for each compiler

Intel (ifort)
FFLAGS = -g -C -ftrapuv -fpe0 -traceback -warn

PGI (pgf90)
FFLAGS = -g -C -Ktrap=fp -Minform=inform

GNU (gfortran)
FFLAGS = -g -fbounds-check -ffpe-trap=zero -finit-real=snan -Wall
Outline

- Introduction to the compilation/Modules environment
- Good practices
- Debugging
  - Why?
  - Basic introduction to gdb, valgrind and TotalView
  - Compilation errors and warning
  - Run time errors and bugs
  - Fortran specificities
  - C/C++ specificities
- Introduction to the optimization
- Validation and sustainability

Floating point exception and incorrect numbers

- Floating Point Exception: modulo 0 \( \Rightarrow \) FPE
- Division by 0 \( \Rightarrow \) Inf
- \( \text{asin}(x), \text{acos}(x) \): \( x \in [0, 1] \Rightarrow \text{Nan} \)
- \( \log(x) \): \( x \in ]-\infty, 0[ \Rightarrow \text{Nan} \)
- \( \text{sqrt}(x) \): \( x \in ]-\infty, 0[ \Rightarrow \text{Nan} \)
- \( \log(y) \): y not initialized \( \Rightarrow \) Bad value

Error of type : double a; a = 1 / 4; Give a=0 and NOT a=0.25

Only modulo 0 generates errors at runtime = FPE

=> Use a debugging tool (gdb or valgrind) to find the error’s line
Segmentation fault

- Array out of bounds
- Dereference a NULL pointer: use a pointer affected to NULL and change it
- Use a pointer to store a value without allocation memory
- Attempting to write read-only memory
- Attempting to access memory the program does not have rights to (the kernel of the OS)

⇒ gcc/g++: -Wall or -Wextra can give some information
⇒ Use gdb with backtrace (bt) or backtrace full (bt full)
⇒ Use valgrind

Wrong result (1)

- **Uninitialized values:** if you use an uninitialized value, the program may not stop and all following calculations will be based on a **random value**
- Static uninitialized variable: no error at runtime
  
  ```c
  double d1, d2; d1 = d2 *10.0;
  ```
- Dynamic uninitialized variable: no error at runtime
  
  ```c
  double d1=malloc(sizeof(double)*10);
  double d2=malloc(sizeof(double)*10);
  d1[3] = d2[4]*10.0;
  ```

⇒ gcc/g++: -Wuninitialized or -Wall will display a warning
  
icec/icpc: -Wuninitialized will display a warning and -g -check=uninit will catch at runtime
⇒ valgrind: « Conditional jump or move depends on uninitialized value(s) »
⇒ gdb: use backtrace
• **Error conversion type: No error at run time**
  – Conversion int -> double/float: Not a problem, C/C++ do it itself!
  – Conversion double/float-> int: Problem
    You have the integer part => Wrong Result!

⇒ **gcc/g++: -Wconversion** informs when an automatic conversion occurs
⇒ **gcc/g++: -Wdouble-promotion** informs when a float is converted in double

• **No change the value of a variable into a function:**
  – Because you don’t use the passage by reference in the function in the call

Example:

```c
void exchange ( int &a, int &b) {
    int c=a;
    a=b; b=c;
}

int main() {
    int i=4, j=6;
    exchange (i,j); // à présent i == 6 et j == 4
    return 0;
}
```

Don’t forget the reference « & » to access at the memory allocated

Without the reference « & » before a and b
⇒ YOU DO NOT CHANGE a and b
Memory leaks

- Not deallocate tab memory
- Reallocation of an already allocated array
- Not close a file in a loop

=> The process stops without warning when the memory is full

⇒ Use valgrind with the option --leak-check=full

1 - Compile with the option -g for gcc/g++ and -g -traceback for icc/icpc
2 - $ valgrind --leak-check=full ./myprog.exe

Error at the compilation

- Argument missing in a call
- Bad declaration of a variable in a call of a function

Example:

```c
Fill_array2(**float psi, int i1,
             int i2, int j1, int j2);

int main()
{
    double **psi;
    int nx = 100, ny=100;
    allocate_double(psi, i1, nx, 1, ny);
    fill_array2(psi, 1, nx, 1, ny);
}
```

Problem with psi => « error: cannot convert ‘double**’ to ‘float**’ for argument ‘1’ to ‘void fill_array2(float**, int, int, int, int)’ ”
Summary for debug (1)

- Depends of the compiler:
  - GNU: gcc for C; g++ for C++
  - Intel: icc for C; icpc for C++

- For GNU: Use some options
  - -Wall: show common warning
  - -Wextra: shows additional warning
  - -Wuninitialized: display warning for uninitialized variable
  - Use the option -g to use the debugger gdb or valgrind

- See man gcc or
  https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Warning-Options.html

An interesting internet site:
- for all warning options
  http://www.nas.nasa.gov/hecc/support/kb/Recommended-Intel-Compiler-Debugging-Options_92.html

Summary for debug (2)

- Intel: Use some options
  - -g: generates symbolic debugging information and line numbers in the object code for debuggers (gdb or valgrind)
  - -traceback: tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at runtime
  - -check=uninit: enables runtime checking for uninitialized variables

Some interesting internet sites:
- http://www.nas.nasa.gov/hecc/support/kb/Recommended-Intel-Compiler-Debugging-Options_92.html
Other way : avoid problems!

- Use the object `<vector>` of the STL (Standard Template Library) to avoid the manipulation of pointer and the risk to have some wrong values or segmentation fault.

- A tab in 2 dimensions is more simple : it’s a pointer of pointer but simple in practice.

- You can use some libraries to manipulate matrix or image as BOOST for manipulation of matrix and OpenCV or ITK for manipulation of image.

Debugging good practices

- compiling with warning and check options
- fix warnings at compilation step
- read the error message at run-time
- use gdb or valgrind if the bug still appears
- use TotalView
- make a break and try again
- ask to your colleague
- ...
Remove debugging options when debugging is achieved

• Check bounds debugging option induces execution up to 10 times longer!!!
• To measure time: `usr/bin/time -p ./a.out`
• As soon as your bug is resolved, remove the debugging options and try optimization options after reading the compiler’s manual (`-O2`, `-O3`, `-fast`, `-Avx`, `-xsse4`, …)

⇒ You are ready for a further training about optimization...

Outline

• Introduction to the compilation/Modules environment
• Good practices
• Debugging
  • Why?
  • Basic introduction to gdb, valgrind and TotalView
  • Compilation errors and warning
  • Run time errors and bugs
  • Fortran specificities
  • C/C++ specificities
• Introduction to the optimization
• Validation and sustainability
Introduction to optimisation

- Evaluate function’s time of your program with a profiler (gprof)
- Have good practices in your development
- Use optimize options of the compiler
- Use openmp for accelerate your loop

gprof

To identify routines where the code spends time.

**Step 1 - Compile your code with -g and -p**
The profiler requires both the debug and profile directives (-g and -p at compilation and link stage)
gcc -c -p -g example.c
gcc -p -g example.o

**Step 2 - Run your program**
./a.out
The profile information is stored in a file called gmon.out.

**Step 3 - Convert gmon.out to a human readable text file**
gprof a.out > timeReport.txt
Some costs

- **Cost of the main operations:**
  - add: 1 CPU cycle
  - sub: 1 CPU cycle
  - mul: 1 CPU cycle
  - div: 20 CPU cycles

- **Cost of the main special functions:**
  - sqrt: very expensive, the number of cycles depends on the input
  - rsqrt: 1 CPU cycle
  - cos: very expensive, the number of cycles depends on the input
  - sin: very expensive, the number of cycles depends on the input
  - tan: very expensive, the number of cycles depends on the input
Good practices

- **Avoid division**: It is better to compute the inverse number and multiply by it! Be careful, when multiply by inverse we lose some precision in the calculation.

- **Avoid branch instructions** (if, switch, etc) or reducing the number of branch instructions.

- **Avoid use pow, sqrt, cos, sin and tan** or reduce the number of calls.

Optimize option with GNU compiler

- **-O0**: reduces compilation time and makes debugging produce the expected results, this is the default.

- **-O1**: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

- **-O2**: optimizes even more, GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff.


Other ways

• Memory access technic
• Cache blocking technic
• In lining (include with the optimization option -O3 of GNU’s compiler)

Validation and sustainability

• Your code gives now accurate results! Congratulations!
• To track sources code, it is necessary to use a version control system (SVN, Git, …)
• To avoid to introduce bugs in new developments, it is really important to define test cases (regression testing) to make sure that the older programming still works with the new changes. It is possible to automate regression testing
  => CPIV
  => JADIM
Lexique

Memory leak : fuite mémoire
Segmentation fault : problème d'accès à la mémoire
Floating point exception : erreur d'arithmétique (ex : division par 0)
Array out of bounds : dépassement de tableaux
Conversion data type : conversion de type de données
Emacs / gdb / addr2line

C Code compiled with –g executed with gdb into emacs

% emacs
   Alt-X
gdb
buggy.exe
run 10
Starting program
/mnt/shared/projects/Cosinus/Shared/Debogage/CPP/buggy.exe 10
SEGFAULT: Reallocation of an already allocated array
Program received signal SIGSEGV, Segmentation fault.
0x0000000000040133c in main (argc=2, argv=0x7fffffd1b8) at buggy_program.cpp: 170

% addr2line --exe=buggy.exe 0x0000000000040133c
/mnt/shared/projects/Cosinus/Shared/Debogage/CPP/buggy_program.cpp: 170

Parallel code: mpiexec -n 2 emacs ...

### Symptom vs Possible causes vs Solution vs Tools to find the bug

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Possible causes</th>
<th>Solution</th>
<th>Tools to find the bug</th>
</tr>
</thead>
</table>
| **SIGSEV** Segmentation fault   | Memory violation due to an array out of bounds                 | Compiler debugging options        | gfortran -g -fbounds-check
                                  | Memory violation due to a missing argument in a routine call, wrong datatype, wrong dimension | Use a debugger if necessary            | ifort -g -traceback -CB
                                  | Stack overflow                                                 |                                   | pgf90 -g -traceback -C
| **FPE** Floating exception      | Incorrect arithmetic                                           | Compiler debugging options        | gfortran -g -fpe-trap=zero,overflow,invalid |
                                  | Uninitialized variable                                          |                                   | ifort -g -fpe-trap=zero,overflow,invalid   |
                                  |                                                               |                                   | pgf90 -g -fpe-trap=zero,overflow,invalid   |
| **SIGILL**, illegal instruction, "instruction non permise" | The executable is compiled for a specific target different of the host | Remove target's dependent compilation options (eg : -arch, -sse4, ...) | gcc/gfortran -g -Wuninitialized
                                  |                                                               |                                   | ifort -g -traceback -C
                                  |                                                               |                                   | pgf90 -g -traceback -Ktrap=fp
| **O**: ALLOCATE: * bytes requested; not enough memory or crash without error message | Memory leak                                                    | Use a debugger                      | valgrind --leak-check=full
|                                                               |                                                               |                                   | TotalView
|                                                               | Resource starvation : lack of memory                           | Check if large arrays are really necessary | Linux command top
| **Supprimé, supprime, abort**   | Swap is full on the computer                                  | Restart the computer to free swap | Linux command df and top
| ** *** glibc detected *** ** | Consistency problem between arguments in the call and the formal arguments declared in the subroutine | Check the dimension and the datatype of the routines arguments | Debugging with break points (gdb, TotalView, ...") to find the source line |