Advanced Methods for Scientific Computing (AMSC)
Lecture title: Compiling, Debugging and Profiling

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

Disabling/reenabling warnings in the source file

Sometimes you may want to disable some warnings on a portion of your code (it happens when using external libraries where the developer has left, for instance, unused variables or comparisons signed-unsigned etc.). In g++ and clang++ (in VisualStudio the technique is similar but not identical), you can selectively disable warnings in parts of your code.
Let assume that foo.cpp is a file where you want to disable the warning `-Wunused-parameter`, which warns you if a parameter of a function is not used inside the function. You can modify the file as follows

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
... // the rest of the file
#pragma GCC diagnostic pop
```

Explanation

Pragmas are compiler directive that are handled by the preprocessor (indeed they start with a #).

`#pragma GCC diagnostic push`

tells the compiler to store the current options related to diagnostic (errors and warnings) on a stack, for later use.

`#pragma GCC diagnostic ignored "<WARNING>"`

tells the compiler to ignore warning `<WARNING>`. The warning is written exactly as the corresponding compiler option. Finally,

`#pragma GCC diagnostic pop`

recovers from the stack the stored settings, so you are evffectively back to the initial situation.

Other useful pragmas to control diagnostic

#pragma GCC diagnostic **warning** "–Wformat"

activates (at warning level) the waring `-Wformat` which tests for conistent use of `printf`.

#pragma GCC diagnostic **error** "–Wnon–virtual–dtor"

considers a missing virtual destructor in a class that has virtual functions at error level. The compilation will stop with an error message (the normal behaviour is a warning.)

A dirty trick

Sometimes you want to silence the "unused variable" warning, without the fuss of creating a **#pragma** GCC diagnostic. In Utilities/hashCombine.hpp you have this piece of code:

```
...
int [] i={0, (hashCombine(seed, rest), 0)...};
//!@note: the following line is to avoid the warning of unused
(void) i;
```

With (**void**) i you cast a variable to **void** and the Standard says: *Any expression can be explicitly converted to type "cv void." The expression value is discarded*.

So now you are "using" i, even if for a useless purpose. The compiler is happy!.

A useful (but not standard) pragma

Header files should contain the header guard to avoid accidental double inclusions:

```
#ifndef HH_A_LONG_UNIQUE_NAME_HH
#define HH_A_LONG_UNIQUE_NAME_HH
// my definitions
#endif
```

A non standard, but simpler alternative, supported by most compilers, is

```
#pragma once
// my definitions
```

Tools for static analysis

There are tools to analyze the code "statically" to find potential bugs. One is cppcheck, which can be integrated into eclipse.

```
cppcheck -I<dir> --enable=all --language=c++ --std=c++17
file.cpp
```

However, it does not detect syntax errors in the code. Cppcheck primarily detects the types of bugs that the compilers normally do not detect.
Ignore messages concerning not found std header files

Tools for static analysis

The clang compiler has the option `--analyze` that tells the compiler to analyze the code instead of producing the object file. Example:

```
clang++ -std=c++20 -I<dir> --analyze ./file.cpp
```

Several IDEs, lamong which Visual Studio Code and CLion can integrate clang++ as a code analyzer.

A C++ interpreter and a C++ explorer

cling is a an interactive C++ interpreter, built on the top of LLVM and Clang libraries.Is part of the ROOT project at CERN. It may be integrated in a jupyter workspace, see here!. It is still in a experimental phase, but having an interpreter may help code prototyping.

With compiler explorer you may check pieces of your code to see how it translates in assembly language, while C++ Insights you see your source code with the eyes of a compiler. See also here.

Optimization options

The compiler is able to optimize the code for better performance: better use of CPU registers, refactoring of expressions, pre-computation of constants etc. However, normally when debugging the code one wants to deactivate optimization.
Moreover, optimization does not always give the expected results. That's why you can modulate the level of optimization through `-On`, where n=0,1,2,s or 3 (other values maybe available, look at the man page).
Option `-O` is equivalent to `-O1` and `-O0` disables optimization.
Option `-Os` (optimize for space) is equivalent to `-O2` but discards optimization that would generate a bigger code. `-O3` is the maximum optimization possible.

Other, more specific, optimization options are available, we mention them when necessary. When optimizing one normally also uses the (preprocessor) option `-DNDEBUG`.

Look at the examples in OptimizationAndProfiling.

Loop unrolling

Often it is better to unroll small loops, i.e convert

```
for (i=0;i<n;++i) // n may be big
  for(k=0;k<3;++k) a[k]+=b[k]*c[i];
```

to

```
for (i=0;i<n;++i){
 a[0]+=b[0]*c[i]; a[1]+=b[1]*c[i]; a[2]+=b[2]*c[i];
 }
```

Compiler may do it for you if you activate the `-funroll-loops` option (but not always you have better performances). It's better to pre-fetch values that are constant inside the loop (but the optimizer will already do it for you).

```
for (i=0;i<n;++i){ auto x=c[i];
a[0]+=b[0]*x; a[1]+=b[1]*x; a[2]+=b[2]*x;
}
```

Try to avoid if inside internal loops!

The if statement is costly. This is terrible:

```
for(int i=0, i<10000; ++i){
  for (int j=1,j<10;++j){
     if(c[j]>0)
        a[i][j]=0;
     else
        a[i][j]=1.;
      }}
```

This looks better:

```
for(int j=0, j<10; ++j){
 if(c[j]>0)
   for(int i=0, i<10000; ++i)a[i][j]=0;
 else
   for(int i=0, i<10000; ++i)a[i][j]=1;
}
```

Avoid std::pow for small positive integer exponents

```
auto y=std::pow{x,3};
auto y=x*x*x;
```

The second solution is more efficient.

Prefer **constexpr** for constant expressions

```
#include <numbers>
const auto pihalf =std::numbers::pi/2.0;//ok but not best
constexpr auto pihalfc =std::numbers::pi/2.0;//better!
```

constexpr is stronger than **const** and should be used when you have constant expressions (i.e. immutable constants). The constants defined in std::numbers are all constant expressions.

An interesting example (for nerds)

CppPerformanceBenchmarks is a suite of benchmarks designed to guide optimization strategies of compilers. You have to compile the tests without optimization to understand the difference of different strategies (and hope that your compiler is able to operate according to the best choice when launched with optimization activated.)

Let's look at one of the many performance tests, concerning the sum of elemnts of a vector:

$$s = \sum_i v_i$$

Two possible strategies

```
double test1()( Iter first, const size_t count ){
double sum(0);
for (size_t j = 0; j < count; ++j)
   sum +=first[j];
return sum;
}
double test2()( Iter first, const size_t count )
{
double sum(0), sum1(0), sum2(0), sum3(0);
size_t j;
for (j = 0; j < (count - 3); j += 4) {
sum  += first[j+0];
sum1 += first[j+1];
sum2 += first[j+2];
sum3 += first[j+3]; }

for (; j < count; ++j)
    sum += first[j];
sum += sum1 + sum2 + sum3;
return sum;
}
```

Compiled without optimization, which is faster? `test1` or `test2`?

The answer

The answer is not easy. It depends on the computer's architecture. On my computer, an `11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz`, `test2` is approximately 10 times faster than `test1`!

Why? The reason is that the Streaming SIMD Extensions (SSE2) instructions set of my CPU allows for parallelization at the microcode level. Indeed, it is a super-scalar architecture with multiple instruction pipelines to execute several instructions concurrently during a clock cycle.
The code of `test2` better exploits this capability.

Lessons to be learned: Counting the number of operations does not necessarily reflect the performance of the code. The compiler optimiser can transform test1 into test2 automatically. Yet, sometimes it's better give it a hand.

Sample-based optimization

If you have to run code many time on "similar" data, i.e. data that will not change greatly the flow of instructions your code will execute, you may activate sampling-based feedback-directed optimization, i.e. optimization based on sampling what the program is doing at run-time.

A detailed explanation goes beyond the scope of this lecture. You may have a look at the documentation of the `-fauto-profile` option of the g++ compiler. In the following slide I just report the content of the documentation

Sample based optimization with g++

Run your code `myprogram` with perf to monitor the execution

```
perf record -e br_inst_retired:near_taken -b -o perf.data \
  -- myprogram
```

Produce the profile data

```
create_gcov --binary=your_program.unstripped --profile=perf.data \
    --gcov=profile.afdo
```

Recompile your code providing the profile data

```
g++ -fauto-profile=profile.afdo -O3 sourcefiles -o myprogram
```

More information on optimization

The g++ options `--help=optimisers` gives a full list of the possible optimization options of the gnu compiler. They are a lot!. If you want to know what is activated when using a generic optimization options of the form `-On`, you can do (here the example for `-O3`)

```
g++ -c -Q -O3 --help=optimizers
```

The optimizations that are still disabled by `-O3` are the "dangerous" ones, i.e. the one that you should activate only when you know what you are doing, or those that may cause an increase of your code size and not an increase in speed, like `-funroll-loops`, or those that may break up IEEE compliance, like `-ffast-math`

Cache friendliness

Often the efficiency of an algorithm depends on how variables are accessed in memory. It is important that in a loop you try to access variables contigously in memory. This allows cache pre-fetching to work effectively. For instance, if mat is a dynamic matrix whose elements are organised row-wise, then

```
for (i=0;i<nrows;++i)
   for (j=0;i<ncols;++i) a += mat(i,j);
```

is cache-friendly. While

```
for (j=0;j<ncols;++j)
  for (i=0;i<nrows;++i) a += mat(i,j);
```

is not cache-friendly, and thus less efficient.

A more complex example is in the MyMat0 folder.

Debugging

When developing a code we want to be able to use the debugger, a program that enables us to execute the program "step-by-step".
To use the debugger the code must be compiled with the option `-g` (and no optimization). The option tells the compiler to add to the source file the information needed to locate to which source line is associated the machine code.
There are two types of debugging:

Static debugging If the code aborts it produced a *core dump*, than can be analyzed. NOTE: you should use the command `ulimit -c unlimited` otherwise the core file is not produced (to save disk space).

Dynamic debugging The execution of the program is done through the debugger. We may break the execution at given points and examine variables.

Compilation with debugger

`g++ -g`. The option must be activated both at compiling and linking stage!.

The most basic debugger is `gdb`. To launch it:

`gdb executable <executable arguments>`

Graphic front-ends are integrated in most IDE (eclipse, CLion etc. A simple graphic front-end is ddd (Data Display Debugger), very essential, but also light. A python based one (looks nice) is gdbgui. A list may be found in this page.

See the examples in OptimizationAndProfiling

Debugging levels

In fact, debugging can be activated at different levels, and it is not forbidden to use also optimization with `-g -O`. `-g` in fact just tell the compiler and linker to produce extra information for the debugger.

However, if you activate optimization you cannot do "line-by-line" debugging in a reliable way, since it may not be anymore possible to make an association between source code instructions and machine code instructions. That's why by default `-g` implies `-O0`.

Sometimes you don't need all level of details `-g` gives, so some other "debugging level" may be useful (but are compiler dependent, here I refer to g++)

Debugging levels and special optimization options linked to debugging

The default debugging level is 2

- `-g0` No debugging information loaded. It overrides `-g`.
- `-g1` Produces minimal information, enough for making backtraces in parts of the program that you don't plan to actually debug.
- `-g3` Includes extra information, such as all the macro definitions present in the program.
- `-Og` Special optimization option. It enables optimizations that do not interfere too much with debugging.

Main commands of the CLI of the gdb debugger

- `run` run the program
- `break` set a breakpoint
- `where` show where we are and all the backtrace.
- `print` show the value of a variable or of an expression
- `list n` show some lines of code around line n
- `next` go to the next instruction, proceeding through subroutine calls
- `step` go to the next instruction, entering called functions
- `cont` continue executing
- `quit` exit the debugger
- `help` As the name says.

Other g++ options

gcc is a cross-compiler, it means that can compile code also for other architectures than the one from which it is run. Moreover there are many options to tailor your code to a specific CPU.

- `-march= [core2|athlo64-sse|amdfam10..]` use the instruction of a specific cpu.
- `-mfpmath=[387|sse..]` Generate floating point arithmetics for selected unit.
- `-msse -mss2 -msse4...` enable SSE extensions
- `-ffast-math`. Faster math operations (but you may loose IEEE compliant arithmetic).
- `-Ofast` Activate optimizations that disregard strict standards compliance (it activates also `-ffast-math`).

The `--verbose` option gives also information on the default architecture (which is deduced automatically).

Other debugging tools

`valgrind` is a very useful tool to find memory leaks, unassigned variables or to check for memory usage.
It requires a lot of memory and the program is very slow when launched through valgrind, since it is run on a "virtual environment".
Find memory leaks

```
valgrind --tool=memcheck --leak-check=yes \
--log-file=file.log executable
```

Check memory usage

```
valgrind --tool=massif --massif-out-file=massif.out \
--demangle=yes executable
ms_print massif.out > massif.txt
```

`massif.txt` is now a text file with indication on memory usage.

The profiler gprof

A profiler is a program that allows to examine the performance of an executable and find possible bottleneck.
Here we give some detail to a very simple profiler that can be used with the g++ compiler. You need to compile the code using the `-pg` option both at compilation and at linking stage.

When you execute the code it produces a file called `gmon.out` which is then used by the profiles

```
gprof --demangle executable > file.txt
```

File `file.txt` will contain useful information about the program execution.
See OptimizationAndProfiling/README

Main options of gprof

`gprof` has a long list of options. The main one (in my opinion). Some options have the `-no-optionName` to deactivate the feature.

- `--annotated-source[=symspec]` Prints annotated source code. If `symspec` is specified, print output only for matching symbols. NOT WORKING ON LATEST KERNELS
- `-I dirs` list of directory to search for source files.
- `--graph[=symspec]` prints the call graph analysis.
- `--demangle` demangles mangled names (essential for c++ programs).
- `--display-unused-functions` As the name says.
- `--line` line-by-line profiling (but maybe better use gcov)

Another profiler: callgrind

callgrind is a tool of valgrind, that you may call, for instance as

```
valgrind --tool=callgrind --callgrind-out-file=grind.out\
--dump-line=yes ./myprog
```

It is suggested (as with other profilers), to compile the program with -g (to load debugger symbols) and optimization activated.
The option `--dump-lines` is here used to have a line-by-line profiling.
In this form, `valgrind` produces a binary file `grind.out` that has to be post processed. The best tool for the purpose is kcachegrind

```
kcachegrind grind.out
```

It opens a nice graphical interface.

Other profilers

As already said there are other profiler tools, some useful also in a parallel environment (gprof is not very good is you do multithreading).

- The command perf: a tool for lightweight CPU profiling.
- The TAU performance system.A portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, Java, Python.
- Scalasca. A performance analysis tailored for parallel applications on large-scale distributed memory systems.

Coverage with gcov

`gcc` compiler supports program coverage with the command `gcov`. A coverage gives you how many times each line of the code has been executed by the program, and it is a useful complement to `gprof` (which gives instead the time spent).

You have to compile your code with `-g -fprofile-arcs -ftest-coverage` and no optimization and, if your code loads a dynamic shared objects with the `dlopen` utility, you need to add at linking stage the option -Wl,–dynamic-list-data.

When you run the code, it will produce files with the name of executable and extensions `gcda` and `gcno`, that will be used by the `gcov` utility.

`gcov` will produce a lot of files, so after having used it's better do a `rm *.gcov`.

Coverage with gcov

```
compile executable with -g -fprofile-arcs -ftest-coverage
gcov [options] source_file_to_examine [or executable]
```

You will have one file for each translation unit used by the file you are examining, with extension `gcov`. They are text files with the code and the indication on how many time each line of code is executed.

Main options of gcov

Also `gcov` has quate a number of options

- `--demangled-names`. Demangle names. Useful for c++
- `--function-summaries` Output summaries for each function in addition to the file level summary.
- `--branch-probabilities` Write branch frequencies to the output file, and write branch summary info to the standard output.

In the example in OptimizationAndProfiling, using `make coverage` will run the coverage of a simple (but not too simple) program.

lcov and genhtml: nice graphical tools for gcov

The output of gcov is verbose and hard to read. But if you install the utility `lcov` (`genhtml` will be installed as well), you can have a much nicer graphical view.

You have to compile your executable using the same rules than for `gcov`, run it, and afterwards do

```
lcov --capture --directory project_dir --output-file cov.info
genhtml cov.info --output-directory html
```

where `project_dir` is the directory where you have the `gcda` and `gcno` files (normally the same of our executable, you can also put `.` if it is the current directory).

In the directory `html` you have the html file, launch your favorite browser on `index.html`

Run `make lcoverage` to have an idea!

Verification and validation

Verification: Is my code doing things right?
Validation: Is my code doing the right thing?

Verification is made during development phase and consists in testing each component of your code (classes, functions...) separately, with specific tests that demonstrates that your code do things right. Make sure that the tests "cover" your code. Make sure you have not memory leaks.

Validation is made on the final code and aims at assessing that it produces the desired outcome: does it converge at the expected rate? does it produce reasonable results that compare well with literature data or experiments? Has the expected computational complexity?

Don't expect testing your code only at the final stage: verify it first component by component.

Early optimization is the root of all evil (D. Knuth)

- Make first sure that you code works by verifying each component separately. During development turn off optimization, activate all asserts, possibly avoid inlining, and use the debugger.
- Identify possible bottlenecks using profilers and, to obtain a better performance, concentrate the "hand made optimizations" there. Don't waste time optimizing a function that accounts only for 0.2% of the execution time.
- Check for memory leaks.
- For the release version, activate all compiler optimizations (and check that the program still works).

Warning: The title DOES NOT authorize you to write scrappy programs! Only avoid early optimizations that require cumbersome data structures or fancy constructs when you should concentrate on making things work.

Non-deterministic errors

A non deterministic error is an error whose outcome may be different on different executions of the program. Sometimes, the error is present only when you optimize the code :(

Non-deterministic errors are rather annoying, but the normal cause is incorrect memory addressing: you are going out of bound in some array, you have a dangling pointer or reference.... Use valgrind, and kill the bug!