Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Compiler and profiling

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

The compiler

- Warnings
- Optimization options
- Loop unrolling
- Aliasing
- How to examine the content of an object file (or a library)
- Static libraries
- Debugging
- Profiling

How to produce an object file

```
compiler other options -c [-DVAR -Iincdir1] \
[-Iincdir2] file.cpp
```

Here, compiler is (in this course) either `g++` or `clang++`. The options `-I<dirname>` and `-D<MACRONAME>` are passed to the preprocessor. In the following we will describe some options used at the compilation stage.

Which version am I using?

`g++ -v` (verbose)
On my computer produces:

```
Using built-in specs.
...
Target: x86_64-linux-gnu
... Thread model: posix
gcc version 4.9.2 ...
```

The `g++ --version` gives a more compact (but less complete) information.

Warnings and errors

The compiler may give errors, which stop the compilation process and warnings.
Warnings are incorrect use of the language which however are not fatal, for instance unused variables. The compiler continues the compilation.
One may select the type of warnings with the option `-W<warninglevel>`. The option `-Wall` activates (almost) all warnings. We suggest strongly to use it.

Option `-pedantic` is "just for pedants" and activates warnings related to programming style.

Which standard?

The most common compilers still assume C++98 standard by default. You may change it by the option `-std=<standard>`, where `<standard>` is either `c++1` or `c++14`.
For example

```
clang++ -std=c++11 myfile.cpp -o myfile
```

Option `-o myfile` tells the compiler the name of the executable to produce (by default is `a.out`). Note that in this case we are also calling the linker, if we want just compilation we use the `-c` option:

```
clang++ -std=c++11 -c myfile.cpp
```

It produces the object file `myfile.o`.

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 --enable=all file.cpp
```

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

```
clang++ -std=c++11 -I. --analyze ./esempio.cpp
```

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, not always the optimization gives the expected results. That's why you can modulate the level of optimization through the option `-On`, where `n` takes the values 0, 1,2,s or 3 (sometimes other values are available, look at the manual of the compiler). Option `-O` is equivalent to `-O1` and `-O0` disables optimization. Option `-Os` (optimize for space) is equivalent to `-O2` but eliminates optimization that would generate a bigger code. `-O3` is the maximal optimization possible.

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

Let's look at the example in OptimizationAndProfiling.

Loop unrolling

Version A:

```
for(int i=0;i<n;++i) for(int j=0;j<3;++j)
y[i]+=a[i][j]*v[j];
```

Version B:

```
for(int i=0;i<n;++i)
y[i]+=a[i][0]*v[0];
y[i]+=a[i][1]*v[1];
y[i]+=a[i][2]*v[2];
```

Version C:

```
for(int i=0;i<n;++i)
y[i]+=a[i][0]*v[0]+a[i][1]*v[1]+a[i][2]*v[2];
```

What is more efficient?

Loop unrolling

Difficult to say in general. Yet it is very likely that version C is more efficient since we avoid the overhead of the `for` loop.

Can the compiler optimize (A) producing (C)? YES, it we authorize it to do so with the following options (g++ only).

- `-funroll-loops` Unroll most internal loops if the dimension is known at compile time (it's a constant expression) and is not too big.
- `-funroll-all-loops` Try to unroll all loops. Beware, it not always produces faster code!!

Aliasing

Code A

```
void fun(Vet const & A, Vet const & B, Vet & R){
...
for(i=0;i<n;++i) for(k=0;k<n;++k)
R[i]+=A[i]+B[k]; }
```

Code B

```
void fun(Mat const & A, Vet const & B, Vet & R){
...
for(i=0;i<n;++i){tmp=A(i); for(k=0;k<n;++k)
R[i]+=tmp+B[k];}}
```

Code B is more efficient since we avoid unnecessary memory access to `A[i]`.
One may think that the compiler could transform A) into B) automatically, but this is not true, why?

Argument aliasing

Because the instruction
`fun(A,B,A);`
is a valid C++ statement: the compiler when compiling `fun()` cannot exclude a-priori that the function cannot be called in that way. The code B) would in this case produce a different result than A)!!
The compiler cannot exclude that the object passed as actual arguments to the function do not overlap in memory, a fact referred to as aliasing.

Two dangerous options

Option `-fargument-noalias` tell the g++ compiler that arguments of a function do not alias each other, while `-fargument-noalias-global` affirms that the arguments do not alias any global variable.

The compiler may then perform more aggressive optimizations. Yet remember that you should maintain your promises. Otherwise you can obtain wrong results.

How to examine the content of an object file (or a library)

A simple example: OptimizationAndProfiling/namemangling.cpp:

```
float pippo(float a){return a*a;}
double pippo(double a){return a*a;}
```

We execute `g++ -c namemangling.cpp`. The command `file` allows to know the type of file:

```
$ file namemangling.o
mangling.o: ELF 64-bit LSB relocatable, x86-64, version
1 (SYSV), not stripped
```

A look at the symbols

In the object file there are symbols (associated to identifiers of variables and functions) which have been defined in the translation unit or merely addressed. In the latter case the linker -or the loader- will try to find the definition in other object files or in the linked libraries.

`nm namemangling.o`

```
0000000000000015 T _Z5pippod
0000000000000000 T _Z5pippof
U _Z9aFunctionRKd
```

`nm --demangle namemangling.o` produces

```
00000015 Tpippo(double)
00000000 Tpippo(float)
UaFunction(double const&)
```

Letter T indicates that the symbol is defined, letter U that it is undefined. The command may be applied to libraries as well. An interesting utility for demangling a symbol is `c++filt`.

Static libraries

In a Unix system a static library is just an archive of object files. The command to create a library is `ar`.

```
ar -rv liblibraryname.a file1.o file2.o .....
ranlib liblibraryname.a
```

The name always starts with `lib` and has extension `.a`. Libraries are normally kept in special directories, usually `/usr/lib`, `/usr/local/lib` (but you may decide otherwise).

Command `ranlib` is not always necessary. It reorganizes the symbols in the library for a faster access.

Shared libraries

We will discuss shared libraries in detail later on. For the time being, consider them as libraries whose loading is deferred until the program is executed. n Unix system they have the extension `.so`. Example: `libumfpack.so`. They are also stored typically in `/lib`, `/usr/lib` and `/usr/local/lib`.
Both static and shared libraries are considered at linking stage to resolve still unresolved symbols. But the code in the static libraries is actually inserted in the final executable.

Linking

In the linking phase the file with the `main` program, possibly together with other object files and libraries are assembled together to produce an executable.

```
g++ <other options> main.o file1.o -Llibdir -lname ....
```

libdir is the name of the directory where to search for libname.a (or libname.so).
All undefined symbols in an object file is searched in the other object files and eventually in the libraries indicated. Note: The order of static libraries matters!

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`. More sophisticated debuggers are integrated in eclipse (but in Linux the front-end is still gdb).

To launch it:

```
gdb executable <executable arguments>
```

See the examples in esempio_errore.cpp

Main commands of the 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 debuggers

There are several graphic interfaces for the debugger: the one of eclipse is particularly complete.

Other programs that provide a graphic interface to the debugger are *ddd*, *Code::Blocks* and *kdevelop*

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`).

Ignorare

Norma

Other debug 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 throgh valgrind, since it is run on a "virtual environment". The amount of information gathered is however great.
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 \newlin
--demangle=yes eseguibile
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. There are several, very powerful one. Beside Valgrind, we mention scalasca for parallel programs.
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 produced a file called `gmon.out` which is then used by the profiles

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

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

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).

- gperftool. Useful to those developing multi-threaded applications in C++ with templates.
- 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.

Early optimization is the root of all evil

- Make first sure that you code works by verifying each component. In the development phase turn off compiler optimization, activate all asserts, avoid inlining, and use the dubugger.
- Then identify the possible bottleneck and concentrate the "user optimization" on those to obtain a better performance.
- For the release version, activate all compiler optimizations (and check that the program works).

Warning: The title DOES NOT authorize you to write scrappy programs! Only to avoid optimizations that require cumbersome data structures when you are concentrating on the algorithm.
Code is written at least twice.