# SOFTWARE AND DEVELOPPEMENT TOOLS PS2 ASSIGNEMENT

PIERRE BLANC FATIN AND NICOLAS PARENT [1]

## CONTENTS



## LIST OF FIGURES



[1] *MSIAM*

float

# 1 INTRODUCTION

In this practical session we implement tests and performance profiles for boost and eigen3, two linear algebra **c++** libraries. The latter aims at comparing performances of the two libraries, we do it for matrix multiplication operation. The former aims at ensuring that the implemented functions of the two libraries are actually executing what we expect from them. For the compilation of such a project, we will use the build system **CMake**.

# 2 COMPILATION AND TESTS WITH **CMAKE**

*We highly recommand the reader to go through the CMakeLists.txt files of our project to get a clearer idea :*
*PS2/CMakeLists.txt*
*PS2/tests/CMakeLists.txt.*

The source we used to learn more about **CMake** is [1]. In order to facilitate the "compilation" of our project we use the build system **CMake**. It permits an easy management of dependencies, include directories, and link editions. It also permits to drive tests.

In order to create an executable **exe**, from some sources we simply gives the list of sources as such :

```
add_executable( exe source1.cpp source1.h source2.cpp source2.h main.cpp)
#second option is to define a variable that contains all sources
set(sources source1.cpp source1.h source2.cpp source2.h main.cpp)
add_executable( exe \${sources})
#third option is to give the path to the headers (relatively to CMAKE_SOURCE_DIR i.e. where the
    CMakeLists.txt is) like this :
add_executable( exe source1.cpp source2.cpp)
target_include_directories( exe PRIVATE path_to_headers )
#PRIVATE means only this target, PUBLIC could be used to transmit it to transmit it to
    dependencies or use
include_directories(path_to_headers)
```

**Algorithm 1:** executable creation

The link edition with (non-standard) libraries is then managed with

```
target_link_libraries( exe \${libraries_to_link} )
#where libraries_to_link is a variable that contains libraries you want to link
#what is usually done with -l{library} and -L{path_to_library}
```

**Algorithm 2:** executable creation

In the case of the **Boost** libraries, the linking is eased with the **CMake module FindBoost**

```
include( FindBoost )
find_package( Boost
  COMPONENTS
    system filesystem unit_test_framework
  REQUIRED
    timer signals
)
#then FindBoost automatically fills some variables relative to Boost :
# Boost_LIBRARIES,  Boost_INCLUDE_DIRS etc ...
```

In the case of **Eigen3** library, it is added as an external project. Since Eigen is a "header-only" library, we only need its directory with the **include_directories** command.

```
include(ExternalProject)
ExternalProject_Add(
```

```
  Eigen
#we suppose here that your project is named TP
#Library Eigen3 is stored in the sub - folder eigen
  SOURCE_DIR  \${Boost_vs_Eigen_SOURCE_DIR}/eigen
  INSTALL_COMMAND echo " Skipping install "
)
include_directories( \${Boost_vs_Eigen_SOURCE_DIR}/eigen )
```

Finally, we add test with the following commands

```
enable_testing()
add_test(NAME name_of_test COMMAND executable_name [argv[1] ...] )
```

# 3 UNIT TESTS

*see code part : PS2/tests for all test files*

In order to ensure that the libraries we implement or use gives right results, we use tests. To do them, we use as explained aforehead **CMake** and Boost.test. For instance we write :

```
BOOST_AUTO_TEST_CASE(first_column)
{
  MatrixXd a = MatrixXd::Random(2,2);
  MatrixXd b = MatrixXd::Zero(2,1);
  MatrixXd c(2,1);
  c = a*b;
  BOOST_CHECK_EQUAL(c(0,0), 0);
  b(0,0)=1;
  // first column
  c = a*b;
  BOOST_CHECK_EQUAL(a(0,0),c(0,0));
  BOOST_CHECK_EQUAL(a(1,0),c(1,0));
}
```

This tests on a very simple example, that the result of a matrix multiplication is right.

# 4 PERFORMANCE PROFILES

For square matrices $A, B$ of size $n$, we compute the time needed to process the matrix product $A \times B$. For Boost and Eigen3, we respectively denote it $\mathtt{time\_boost}(n)$ and $\mathtt{time\_eigen}(n)$. Note that those times can be seen as random variables, but it happens that those times are quite steady if we repeat the computation $A \times B$ for many $A, B$ random matrices of given size. Anyway, we have averaged those times on a few repetition of the operation.

We also modify the compile options thanks to the following **CMake** command

```
    add_compile_options( options )
    # or to use it only on one executable (and its dependencies if PUBLIC)
    target_compile_options(target_name PRIVATE options )

```

where options are taken among :

- -g
- -O0 : no optimization
- -O1 : speed up
- -O2 : O1 + loop unrolling

- -O3 : transform loops + memory access speed up

Let's start be comparing the performance without compilation optimisation of both library. The following graph is the result of our matrix multiplication benchmark in function of matrix sizes.

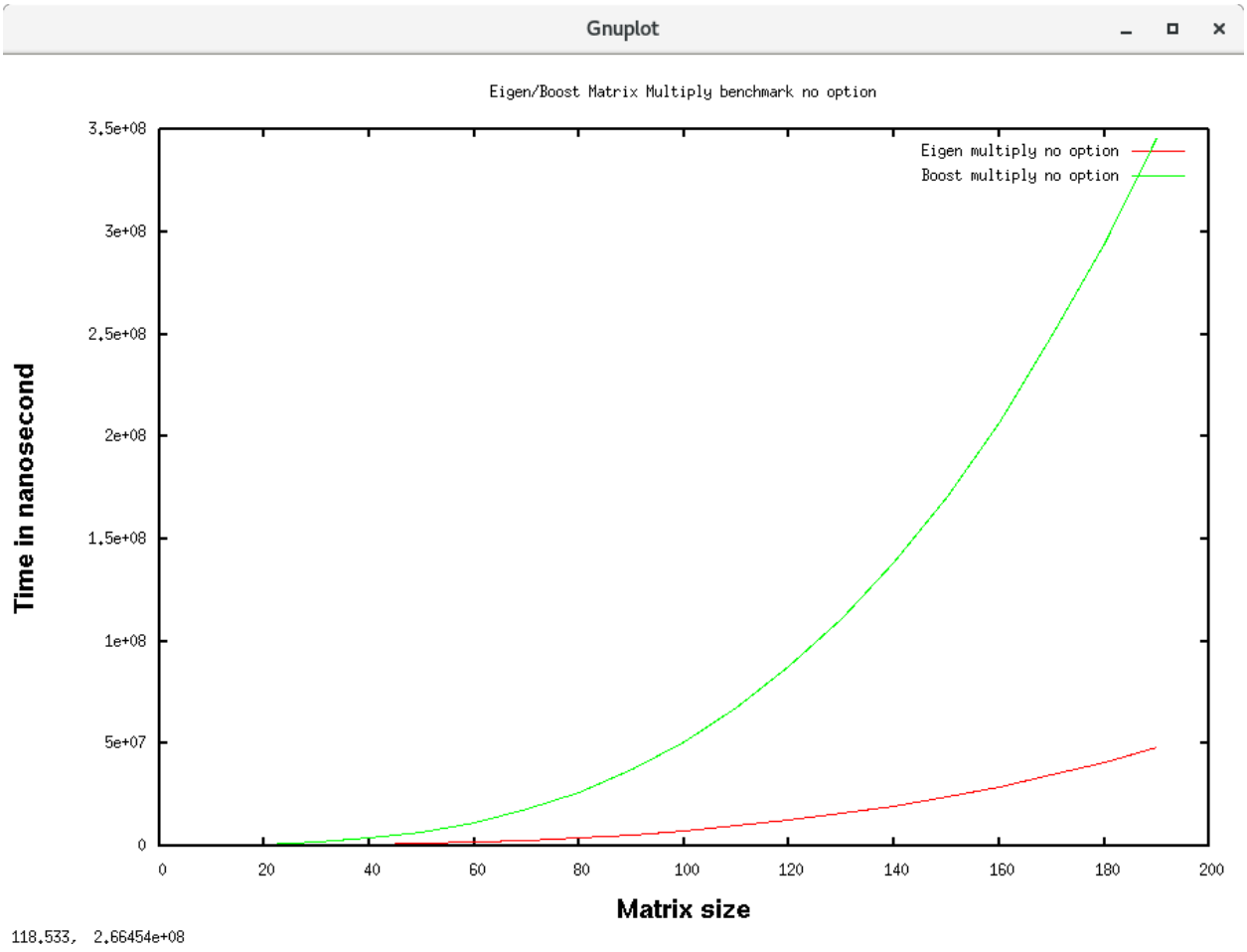


**Figure 1:** Matrix multiply benchmark with no compile options

One can graphicaly see that Eigen performance is a lot better than Boost about matrix multiplication, even with a small matrix size. This is shown here with no compile options but has also been checked with the aforehead mentionned options.

Now let's see what is the effect of the different compile options one both library with 2 and 3

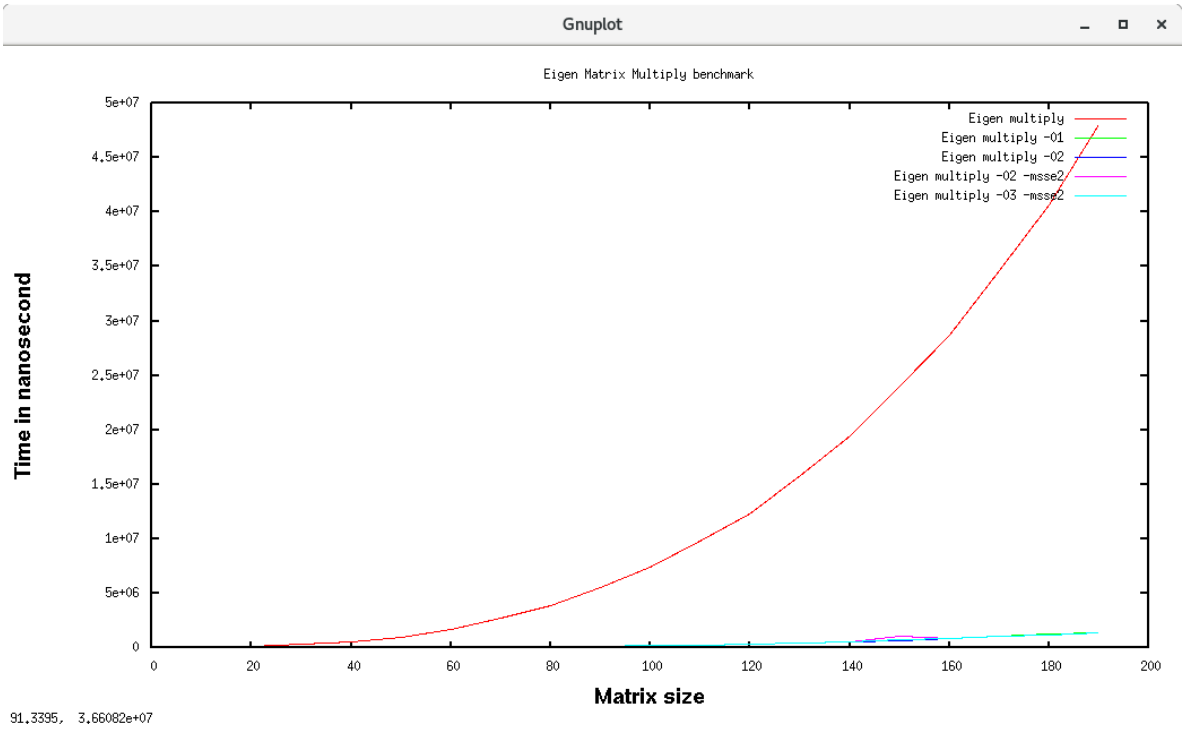


**Figure 2:** Eigen Matrix multiply benchmark with different compilation options

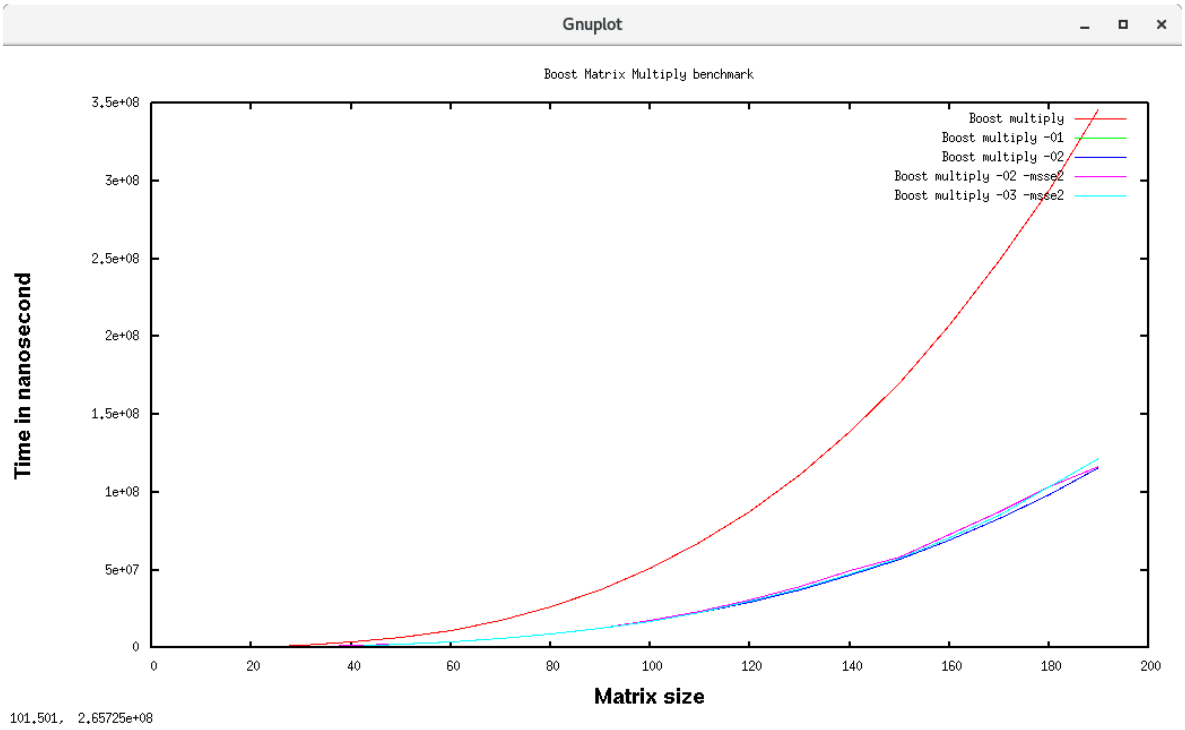


**Figure 3:** Boost Matrix multiply benchmark with different compilation options

One can observe for both library a great performance improvement between without (red line) or with (other color lines) compile options. But the performance gain over all different levels of optimisation during compilation seems negligible. Note also that the compilation optimisation give much better result for Eigen than for Boost. The choice of doing further optimisations during compilation is a trade-off. The higher the optimisation level is, the slower will be the compilation, but the faster will be the execution.

Now we will be interested in affecting multiple cpu to the matrix multiplication task. Our lab computer have 4 cpu, so we will affect 4 cpu to the multiplication task. We do this using the command 'taskset -c 0,1,2,3 ./matrixMultiplication'. In figure 4 we find performance comparison using 1 or 4 on the eigen matrix multiplication. Notice that for this plot we have used the "-O3 -msse2" compile options for both curve.

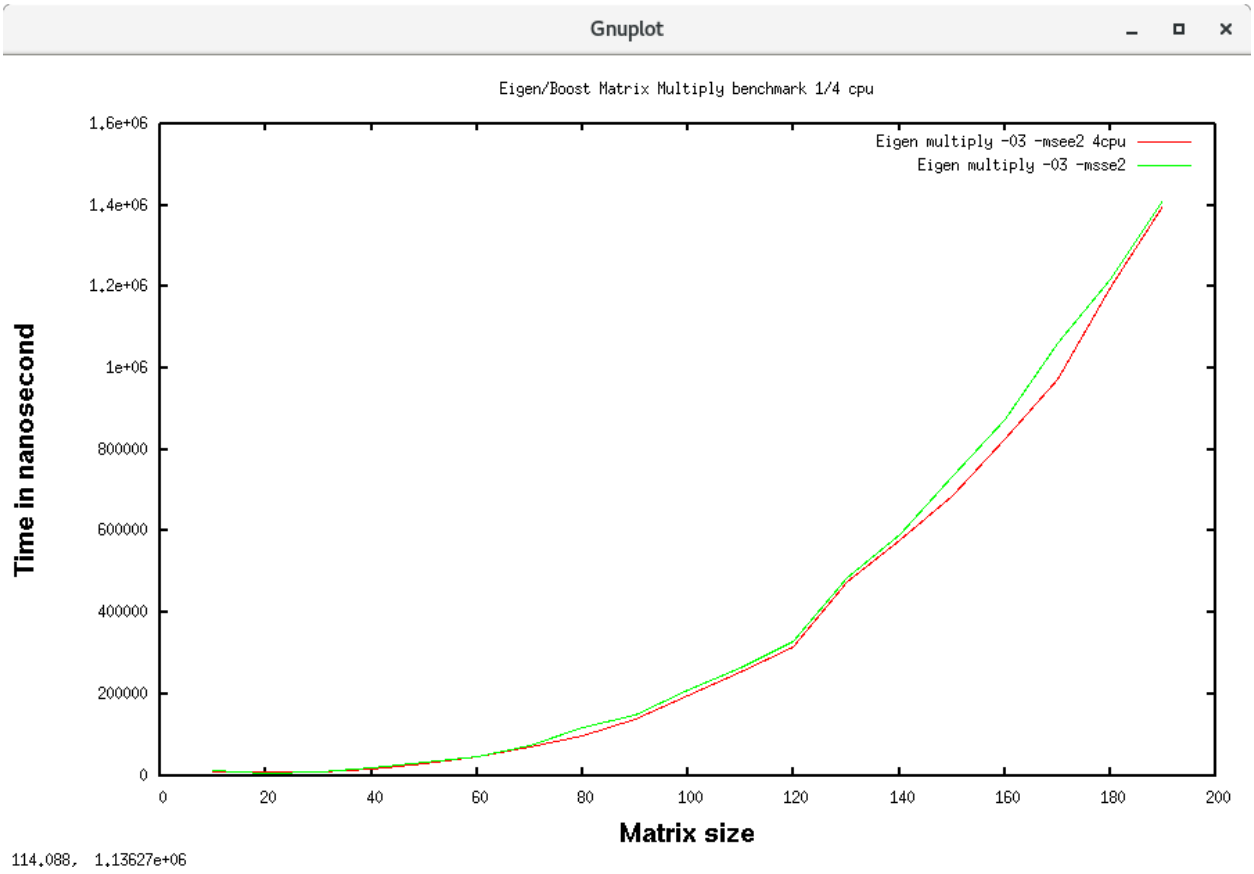


**Figure 4:** Eigen Matrix multiply benchmark using 1 and 4 cpu

One can notice that the time improvement of the 4 cpu affectation is very low for this example. It's maybe due to the fact that the eigen matrix multiplication is not parallelised. Therefore one single thread doing the computation on one cpu. So there is no point to affect more than one cpu to this task.

# 5 CONCLUSION

This practical gave us the chance to improve our knowledge on **CMake** and see how it facilitates the compilation and linking process by creating a Makefile for us. In the comparison of the two libraries, Eigen clearly surpass Boost. More important is the effect of the compile options, which at the cost of a slightly longer compilation, brings huge improvement in the matrix multiplication operations. A continuation track would be to compare the performance of the two libraries to solve linear systems (at least in mean square).

## REFERENCES

[1] Pérignon and al. Outils developpement, 2019. URL https://pole-calcul-formation.gricad-pages.univ-grenoble-alpes.fr/ced/outils_devel.