Teratec Hackathon 2025

Team - Exafloppeurs

Alexis LABILLE - Kevin LAMY – Kilyan SLOWINSKI

Sommaire

Optimizing the Viridien Code

Git and content

The project content is available in the following git : https://github.com/iAmako/Teratec2025-TeamExafloppeur

Its content and usage is described in the README markdown file.

First Benchmarks

We start by executing the original program and profiling it to get a first measure and understand where we can get our first speed-ups.

Main Loop

For an unknown reason, the black scholes monte carlo function is called twice for each run to compute theoretical and actual prices, in the two cases the parameters are the exact same resulting in a quite random error for each run. We removed this part of the loop since it didn't seem to have a use in the result which ultimately divides the total time for long by roughly 50%.

Floating-Point Arithmetic

The original BSM code uses double-precision floating-point format for most computations. By changing to single-precision floating-point format we get the same end result in less computing time. This results in a 25% time reduction with no major distortion to the result.

Parallelism

Monte Carlo algorithms are ideally suited to parallelism programming. The cluster used has 2 multicore CPUs with 96 and 64 cores and no GPU. Therefore, we decided to parallelize the code using MPI. We divide the total work to be done by the number of CPU cores and send them the necessary data. The resulting sums are reduced into a single value at the master process.

To launch our programs using MPI we use the mpirun command as shown in our multiple test scripts. This command is used on the front machine as we give it an hostfile with the target machine(s) and the number of processes to use. In the case of the c8g machine we make our test with 96 processes while we use 64 processes on the c7g. Later, we use an hostfile that contains the 2 nodes and launch 160 processes to get better times.

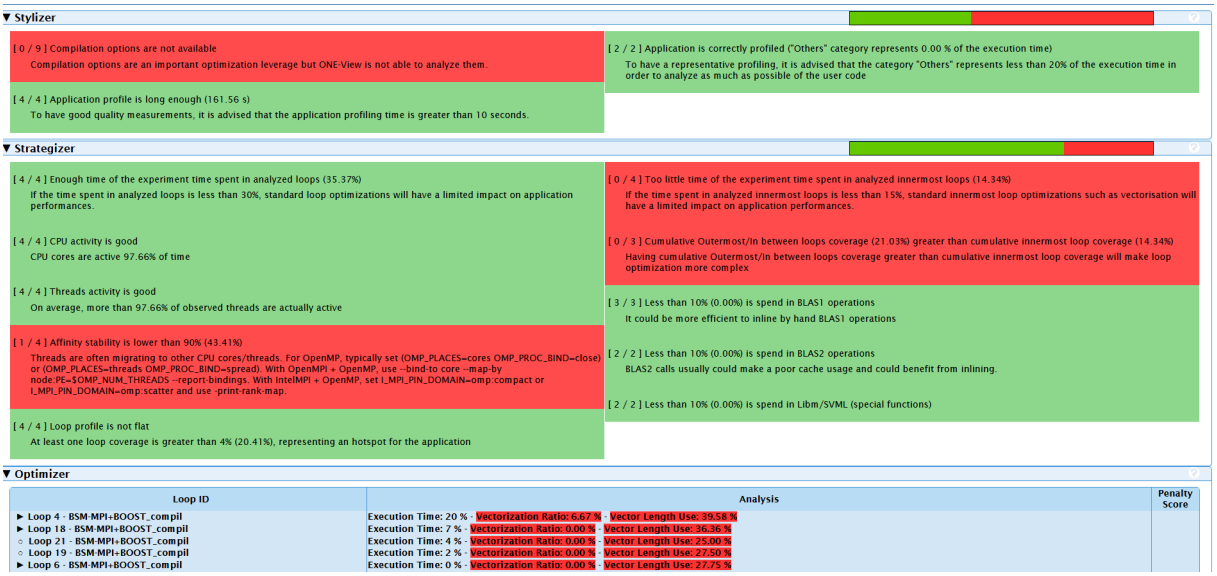
Specifically, this is the command we use in the latest execution, assuming to be at the root of the project : `mpirun --hostfiles/all_hosts -n 160 exe_files/filename first_parameter 1000000 > log_files/filename`

With no surprise this is our biggest speed-up, to a point where we can't really make a comparison as the non-parallelized program is too long to get a time with high parameters and the parallelized program with low parameters result in times that are polluted by the resource allocation which isn't relevant on the target parameters.

Another way to parallelize the code that may get better results would be to have MPI divide the work in the cluster and OpenMP.

Vectorization & MAQAO benchmark

We conducted several MAQAO benchmarks in our experiments. On the latest code, MAQAO shows that vectorization is our main bottleneck.



MAQAO makes it clear that the affinity stability is too low and may get in our way of optimizing the code.

Random Number management

The original random number generator used is the standard library `sdf:mt19937` with `std::normal_distribution`. We replace those with the number generation from the Boost C++ library that is known to get good results in high-performance computing contexts.

Compilation

To compile the source files into executables we use `MPICXX`. `MPICXX` invokes g++ by default to compile the source files, since we're using graviton3 & graviton4 arm architectures we also tried to compile the code with `MPICXX` and `armslurm++` as suggested by the official arm documentation. This has shown no speed-up, therefore, we backtracked to using `g++`.

Performance for each compiler with parameters : 10 000 000 / 10

armclang++	1.71 seconds
g++	0.354 seconds

Since we're working with a lot of floating-point numbers that do not need an extreme precision we use `-ffast-math`. We also tested out every Ox variation, while O3 gets worse times than even O1 we get a better speed-up with the O2 compiling option. We also tried the `march` & `target_cpu` options that resulted in no speed-up, showing the efficiency of base gcc.

The main compilation line we now use is : `mpicxx -o exe_file -ffast-math -std=c++17 -lmpi -O2 cpp_file`

Performance difference for compilation options with parameters : 1000000 / 1000000

without args	749.079426 seconds
with optimized args	126.454629

Ultimately, arguments get us a speed of nearly 6 times.

Performance measurement and validation

Performance measurements are done by using multiple scripts available in the `/scripts` folder- we used these scripts so that multiple tests can be done at night during our resting time. The results are registered in the `/logs` folder.

Here is a summary of times obtained :

Architectural difference between Graviton3 & Graviton4 with 10 000 000 / 1 000 000

Graviton3 - 64 processes	3086.213066 seconds
Graviton4 - 64 processes	2526.409786 seconds
Graviton4 - 96 processes	1682.320837 seconds
Graviton3 + Graviton4 - 160 processes	12335.59192 seconds

Best times, with optimizations

Parameters	Time (In seconds)
100000 / 1000000	14.931379
1000000 / 1000000	126.391859
10000000 / 1000000	1290.454791
100000000 / 1000000	12335.591923

Conclusion

Currently, our biggest and most efficient run is with the “100 100 100” /. “1 000 000” parameters, using all the optimization above in the “`log_files/output_BSM-MPI+BOOST_all_hosts_160.txt`” file in roughly 3.5 hours.

While we could have made a test with “1 000 000 000” - “1 000 000” in about 30 hours we decided to stop it 10 hours in to benchmark the program and find potential bottlenecks. Currently, the biggest remaining bottleneck seems to be the vectorization part.

Code Aster

Presentation

Code Aster is a software designed for numerical simulations in the fields of mechanics and engineering, developed by EDF.

It is mainly used to perform simulations based on the finite element method to study stress, deformation, vibrations and more.

Since Code Aster is open-source, it is free and can be easily modified by users.

Installations

Singularity

According to the installation documentation on Linux, Code Aster is generally paired with Salomé-Meca, a graphical interface for modeling and post-processing. However, it recommends running inside a container via Singularity and, since the provided images are compiled for x86-64, they cannot be used on Arm64.The first step would be to compile an image directly on the cluster to make it compatible and run the container without any issues.

A section of the documentation explains how to install Singularity. For an unknown reason, the default installation on the cluster didn't work, so it was necessary to compile it from scratch. The basic prerequisites for Singularity are installed with the following command :

```
sudo yum update -y && sudo yum groupinstall -y 'Development Tools' && sudo yum install -y openssl-devel libuuid-devel libseccomp-devel wget squashfs-tools
```

Another prerequisite is Go, the programming language supported by Google. It can be downloaded and installed with these commands :

```
wget https://golang.org/dl/go1.23.5.linux-arm64.tar.gz
```

```
tar -C ./ -xzf go1.23.5.linux-arm64.tar.gz
```

Then, the folder path must then be added to the environment variables :

```
echo 'export GOPATH=${HOME}/go' >> ~/.bashrc
```

```
echo 'export PATH=/usr/local/go/bin:${PATH}:${GOPATH}/bin' >> ~/.bashrc
```

```
source ~/.bashrc
```

If everything went well, running `go version` should execute normally.

Singularity will then be installed using the following commands, which will be confirmed by `singularity version` :

```
mkdir $HOME/dev && cd $HOME/dev
```

```
git clone https://github.com/sylabs/singularity.git && cd
singularity

./mconfig

cd ./builddir && make

sudo make install
```

Once this is done, the prerequisites for Code Aster need to be retrieved (`cd ../.. && wget https://www.code-aster.org/FICHIERS/prerequisites/codeaster-prerequisites-20240327-oss.tar.gz && tar -xvf codeaster-prerequisites-20240327-oss.tar.gz`)

Different container files are available in the extracted folder (`cd codeaster-prerequisites-20240327-oss && cd container`). However, to run them, a web link inside the files need to be removed, as it leads to a 404 error at the time of writing. If this link is removed, the compilation process fails due to missing prerequisites. It seems these prerequisites are downloaded specifically from the removed link and are therefore unavailable. As a result, the idea of using Singularity was abandoned, with plans to revisit it later if time allowed.

From scratch

The next step is to compile Code Aster from scratch, which can be done with this command : `mkdir -p $HOME/dev/codeaster && cd $HOME/dev/codeaster && git clone https://gitlab.com/codeaster/src.git`.

Once the compilation folder is downloaded (`cd src`), it is necessary to check if the prerequisites are met using `./waf configure`.

This command will indicate which libraries are required for the installation configuration and will fail if any are missing. Specifically, four libraries are essential to pass the configuration : HDF5, Med, METIS and ParMETIS. Each of these must be downloaded, compiled and installed.

HDF5 is an open source framework for storing and managing large datasets. The 1.10.2 version must be used to maintain compatibility with Med. It can be retrieved with the following command : `wget https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.10/hdf5-1.10.2/src/hdf5-1.10.2.tar.gz && cd hdf5-1.10.2 && mkdir build && cd build`.

The command `make config prefix='/usr/local/hdf5' && make` is used to provide the installation folder and compile the files before running `sudo make install` to install.

To retrieve Med (an open-source framework designed for handling scientific and engineering data), `wget https://files.salome-platform.org/Salome/medfile/med-4.1.1.tar.gz && cd med-4.1.1 && mkdir build && cd build` is retrieved before being configured by the command `../configure -prefix='/usr/local/med' --with-hdf5='/usr/local/hdf5' --with-swig=yes`.

These options specify the location of HDF5 (which will cause the installation to fail if a version higher than 1.10.2 is used) as well as the use of SWIG, a library that enables wrapping C/C++ code for Python.

`make -j $(nproc) && sudo make install` allows to compile and install the library. The `-j $(nproc)` option simply speeds up the installation process by using all available cores.

Next, there is ParMETIS, which requires the installation of METIS, and METIS requires GKlib. Each of these libraries is installed in the same way as the others (GKlib (a library of various helper routines and frameworks used by many of the lab's software) : `git clone https://github.com/KarypisLab/GKlib.git`, METIS (a set of serial programs for partitioning graphs, finite element meshes and producing orderings for sparse matrices) : `git clone https://github.com/KarypisLab/METIS.git` et ParMETIS (a MPI-based library): `git clone https://github.com/KarypisLab/ParMETIS.git`).

```
mkdir build && cd build && make config prefix='/usr/local/metis'

make && sudo make install
```

Finally, it is necessary to add the path to each library in both PATH and LD_LIBRARY_PATH so that the system can access the required resources.

```
echo "export PATH=/usr/local/hdf5/bin:\$PATH" >> ~/.bashrc && echo "export PATH=/usr/local/med/bin:\$PATH" >> ~/.bashrc && echo "export PATH=/usr/local/metis/bin:\$PATH" >> ~/.bashrc

echo "export LD_LIBRARY_PATH=/usr/local/hdf5/lib:\$LD_LIBRARY_PATH" >> ~/.bashrc && echo "export LD_LIBRARY_PATH=/usr/local/med/lib:\$LD_LIBRARY_PATH" >> ~/.bashrc && echo "export LD_LIBRARY_PATH=/usr/local/metis/lib:\$LD_LIBRARY_PATH" >> ~/.bashrc

source ~/.bashrc
```

Once this is done, and after returning to the Code Aster installation folder, the configuration command is updated to specify the locations of each library and some tags. It become `CPPFLAGS='-I/usr/local/hdf5/include -I/usr/local/med/include -I/usr/local/metis/include' LDFLAGS='-L/usr/local/hdf5/lib -L/usr/local/med/lib -L/usr/local/metis/lib' ./waf configure`

`--prefix=/opt/aster/install/mpi --hdf5-libs='hdf5' --med-libs='med' --metis-libs='metis' --disable-mumps`

`–disable-mumps` allows the configuration to proceed without MUMPS, enabling a basic installation with only the mandatory libraries. Once the installation is successful, adding the missing elements that were not found during the configuration will resolve the different ‘not found’ errors.

Before using this command, it will be necessary to modify the GCC version. Indeed, the version installed by default on the cluster is 6.5, while the version required during the installation process will be 14.2. This version is already present on the machine, it just need to be loaded using the following command : `module use /tools/acfl/24.10/modulefiles/ && module load gnu/14.2.0`

If all the installations proceeded normally, the configuration will complete without any issues and will allow for a successful installation using this command :

`CPPFLAGS='-I/usr/local/hdf5/include -I/usr/local/med/include -I/usr/local/metis/include' LDFLAGS='-L/usr/local/hdf5/lib -L/usr/local/med/lib -L/usr/local/metis/lib' ./waf install -j $(nproc) --prefix=~/dev/code_aster --hdf5-libs='hdf5' --med-libs='med' --metis-libs='metis'`.

At this point, various issues were encountered :

- Some files in Med were not recognized by each other (medfile.h -> med.h) : it was necessary to add the corresponding includes.

-In Med, a function (`MedparOpenFile`) was not enabled, even though it was present in the code. Despite various searches, there was no way to enable it normally. It was only enabled under the definition of a constant (`MED_HAVE_MPI`). The file containing the function definition was modified at the beginning (`#define MED_HAVE_MPI 1`).

-Still in Med, it was necessary to modify the size of the INT directly in the Makefile. It was later discovered that it was enough to add the flag `-i64=1` during the `make config` step.

Once these issues were resolved, the compilation proceeded normally and the installation began. However, a new problem appeared : some functions were not recognized despite being present in the original source code (such as `MedparOpenFile` or `gk_errexit`). No solution was found before the clusters were shut down. However, after reading the documentation once again, this could be due to the `–with-swig=yes` option during the installation of Med. Indeed, since it is a wrapper library but no actual path to the library was provided, the configuration likely set a default link that points to nothing.

As a result, Code Aster could not be installed on our cluster, probably due to a simple error flag.