Teratec Hackathon Report

Team Shrek

Team Members

Tanguy Chatelain
chatelain.tanguy@lilo.org

Abdurahman Kanit
kanitbensaidi.ac@gmail.com

Louis Peyrondet
louis.peyrondet@hotmail.fr

Tristan Riehs
tristan.riehs@bordeaux-inp.fr

Professor

Mathieu Faverge
mathieu.faverge@inria.fr

January 27, 2025

Contents

1 Foreword

Our work is available at `https://github.com/Tanguy-chtln/hackaton-teratec-2025`.

2 Viridien

The first project that will be tackled in this report is from Viridien. It is a simulation of the evolution of the price a given asset using the Black-Scholes model. This model is solved using a Monte-Carlo-based method. In a nutshell, it consists in using probabilities to get as close as possible to the wanted value by exploring numerous—billions of—scenarios, thereafter called *branches*.

To evaluate performance, we will use the *Branch Per Second* (branch/s) metric. Traditional HPC benchmarks use the FLoating-Point OPeration per second (flop/s), but our simulation not being pure linear algebra, it contains procedures such as exponentials and Random Number Generation (RNG) that are difficult to translate in terms of number of operations. Unless told otherwise, every benchmark in this section were made on the Graviton 4 CPU.

Landmark 1 The given code had a performance of 23Mbranch/s.

2.1 Given Code

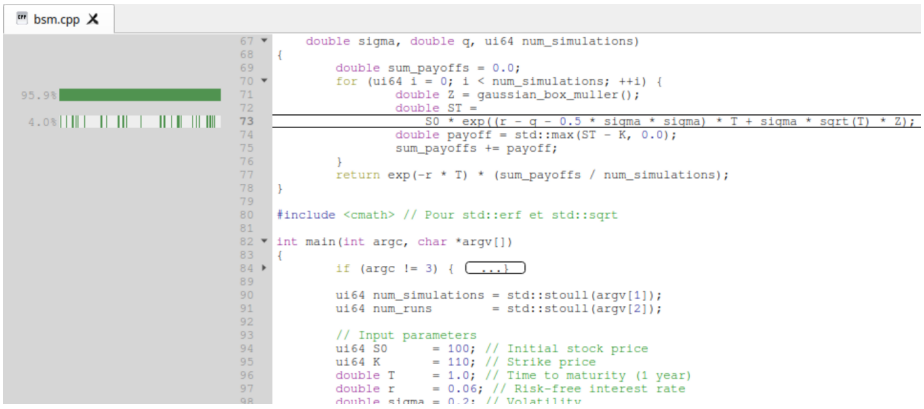

Figure 1: Profiling of base code. This profiling was carried out with Linaro Map using 20,000,000 simulations, and 100 runs.

Before editing any code, we compiled it as-is and profiled it. In the beginning of the hackathon, we were given an introduction about Linaro Forge, a set of tools for—among other things—debugging and profiling parallel programs. We

used it and found out that more than 90% of the execution time was spent doing RNG.

Figure 1 shows the experiment, where on the left side there is the total execution time taken by some line of code, in this case 95.9% of the time doing the `gaussian_box_muller` generation and 4% in the `exp` function.

Looking into the time spent in the RNG, most of the time is taken by the normal distribution function which calls `logl` and `divtf3` for 88% of the time. Linaro Map also provides many other kinds of metrics on the execution such as, but not limited to, branch mispredicts, cache misses, context switches and thread activities.

2.2 Random Number Generation

The vanilla given code used two functions of the C++ standard library in order to perform RNG following a Gaussian distribution. One function per step of the process:

1. generate a random number uniformly;
2. interpolate that uniform distibution to a Gaussian one, done using the Box-Muller transform in the original code.

2.2.1 Using the ARMPL RNG

We recalled that most modern architectures have a hardware support for RNG. The Graviton 3 and 4 have an `rng` flag[1], which indicates that this kind of feature is available. Thus we looked for an API in the ARM Performance Library (ARMPL), pre-installed on the cluster, and found the *OpenRNG* API.

This API provides several types of RNG, and does not abstract the two aforementioned steps.

First, it exposes an opaque datatype that is a *stream*[2] of random numbers. A stream generates uniformly distributed numbers. Several ones are available and the algorithms they use are well documented, using appropriate references to scientific papers. They all use a seed and a simple mathematical relation to compute the stream of numbers. For example, a *multiplicative congruential*[3] approach would use a relation of the following form. The constant values a and m typically have more than 12 digits, and R_0 is the seed.

$$R_{n+1} = aR_n \text{ modulo } m$$

Then, using this stream, we can provide a buffer as well as an interpolation method to get our Gaussian RNG[4]

[1] Information about the CPU can be retrieved from the `/proc/cpuinfo` file on Linux systems.

[2] *Stream* is part of the actual name of the datatype.

[3] Quoting from the ARMPL documentation.

[4] In fact, numerous usual distributions are implemented in the ARMPL.

This implementation does not use any hardware-specific feature that generates random numbers. This type of device is generally tailored for cryptography, not for HPC. That means that the emphasis is on the quality and the unpredictibility of the RNG rather than the throughtput. In HPC, even if the RNG distribution quality plummets, having a significantly higher thoughput may be an interesting tradeoff.

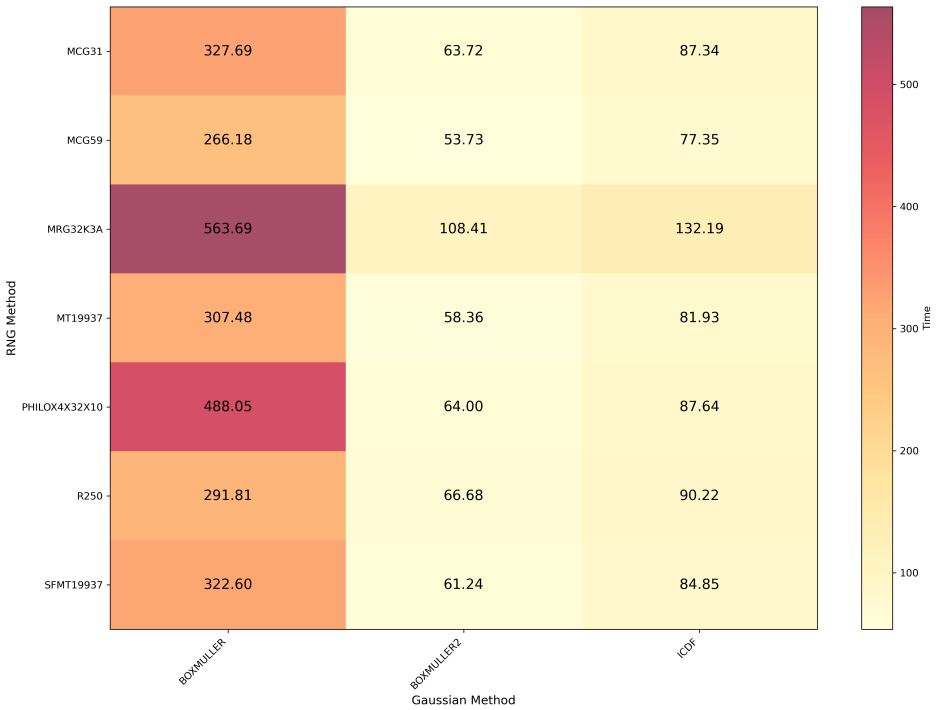

Figure 2: Performance of different RNG algorithms. These performances correspond to a single-threaded execution with 10,000,000 simulations and 1,000 runs.

We wrote a script that compiles a version of the program for each (uniform generator, Gaussian interpolator) pair—27 in total—and runs them in order to have an idea of which ones perform the best. Figure 12 shows the results we obtained.

The figure does not contain 27 cells, two uniform RNG methods have been omitted. One was removed because its execution time exceeded 2,000s. Having included it would have undermined the readability of the heatmap. Another one was omitted because the experiment returned "inf" as a value.

From this point, we always used the multiplicative congruential MCG59 uniform generator, coupled with the Box-Muller-2 transform.

Landmark 2 Using ARMPL RNG, the new performance is 186Mbranch/s. The speedup from this first optimization is 8.

2.2.2 Optimal Buffer Size

In the beginning, the original program used to call the RNG function for each iteration, that is, $N_{\text{simulations}} \times N_{\text{runs}}$ times. Without any knowledge regarding how the standard C++ library RNG is implemented, we cannot assert that there was absolutely no buffering, and that expensive computing was performed on every single call to the RNG routine. But since two class constructors were called, we believe no buffering was done.

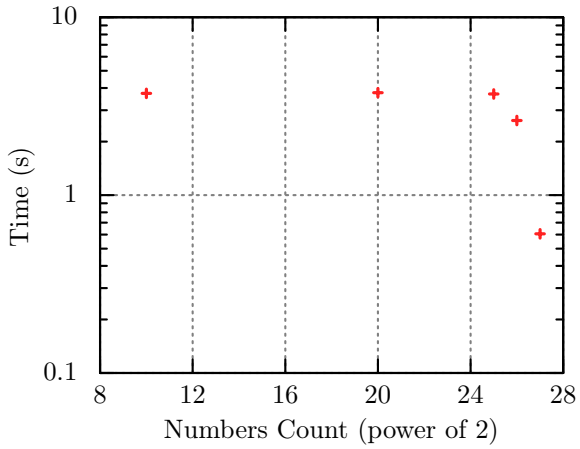

Figure 3: Performance of different random numbers buffer sizes. The experiment was carried out using 10,000,000 simulations, 1,000 runs, and 96 threads.
Note: muti-threading will be tackled in section 2.4.

When we implemented our RNG using ARMPL, we used buffering. At first, without thinking too much, we thought of generating random numbers by batches of size $N_{\text{simulations}}$. Then, we wondered what buffer size would be the best for RNG.

Figure 3 shows the performance of a few different sizes. We clearly observe that the best one is 2^{27} numbers, or approximately 134 million numbers. Since we used 8-byte double precision numbers, the whole random numbers buffer's size was 1GB. In fact, this is the highest we could measure. The program would output an execution time of 0 with a twice as big buffer. We did not invest time in troubleshooting it.

Unfortunately, since we did this benchmark after adding multi-threading (again, this will be described in section 2.4), we cannot really compare the performance of this optimization with landmark 2. If we divide the performance

by 96, the number of threads, we get 172Mbranch/s, which is less than the 186Mbranch/s we measured earlier, where the buffer size was $N_{\text{simulations}}$ numbers, which was equal to 10,000,000.

We can clearly see on figure 3 that making the buffer bigger improves performance. In fact the performance loss in terms of branch/s between landmark 2 and now comes from the fact that CPU frequency decreases when the number of used cores becomes high. Please be patient, section 2.4 will further analyze this behavior.

2.2.3 Understanding the Implementation

We are now using the ARMPL RNG, but could there be an even faster alternative? What guarentees that the ARMPL RNG is the fastest option?

This is a complex question, and RNG is a broad topic we are not experts in. However, we wanted to at least verify that it uses vectorization. The program does not do any interaction with devices that have a low memory bandwidth such as a hard disk, a network, or—to a lesser extent—the RAM. It is hence completely compute-bound, using vectorisation is crucial in this type of scenario.

The ARMPL is not open-source, which means we cannot access its implementation of the OpenRNG API we are using. We therefore had no choice but to disassemble it. We disassembled the `vdRngGaussian` symbol from `libarmpl.so` using `objdump`. It is the function we use to generate our random numbers.

In fact, the `vdRngGaussian` symbol disassembly contained only error checking and a call to another procedure that has a 91-character name we omit. But the disassembly of the final procedure mainly consisted of vector intructions. Vectorization will be further analyzed in section 2.6.

The conclusion of this investigation is not that we can state this RNG approach is the fastest we could have used, but that it is quite unlikely that by implementing RNG algorithm using vectorization ourselves would be any better. At this point, we were fine with RNG and moved on to other optimizations.

2.3 Precision

In order to be able to evaluate the correctness of our optimizations. In the first night of the hackathon, we ran the longest possible simulation with the given program to gather data that is guaranteed to be correct.

However, the original program being slow, we cannot execute it over 100 million simulations and 1 million runs, that would take days. We used only the original program to test the correctness of our optimized versions on smaller executions.

Finally, what we use as a reference for precision is the result of an optimized program that uses double precision. We need to use this as a reference because the original program can only converge to a 3-decimal value. If we want to compare more digits, we must resort to an optimized version that is able to converge further.

Precision	Value	Time (s)	Performance (Gbranch/s)	Correct Decimals
double	5.136955	1304	7.67	5
simple	5.135329	757	13.2	2

Table 1: Summary of performance of simple and double precision. Experiment carried out with 100,000,000 simulations and 100,000 runs, using 96 threads.

Tables 1 shows that using simple precision instead of double increases performance by 72%. The maximum speedup brought by using simple precision being 100%.

We wanted to do this benchmark with 1,000,000 simulations in order to see how simple precision performs on the largest execution we can run. Unfortunately during the last night of the hackathon, when the job was running, our cluster died due to the disk being full.

Looking back at it, it would have been better to compare two executions that lasted the same time to see which one was more precise. Unfortunately, we did not think of this idea back then.

Since results for simple precision were not satisfying in the current state of our work. We decided not to try using half precision.

We had looked for this feature in the beginning of the hackathon. We originally planned to try it, but wanted to test simple precision beforehand. Even though we found out that Graviton 4 has hardware support for half precision SIMD computing, there are problems regarding our code.

The RNG API as well as the vectorized exponential do not support half precision. That means we would have to implement these features ourselves in order to use half precision. This is not impossible to do, but in the context of a 5-day hackathon, we decided not to dive into it.

2.4 Multithreading

We added an OpenMP layer to our code in order to make it parallel and benefit from all the 96 cores of the Graviton 4 processor. The computation performed in the code does not involve a lot of memory movements. It is very much about generating random numbers, applying a function, and performing a reduction to sum all the values. Each thread can independently generate its own data and manipulate it. Moreover, the ARMPL RNG implementation is pure which is perfectly suitable for being used in a parallel program.

Figure 4 shows the weak scalability of the resulting parallel program. In this benchmark, the amount of calculation to perform *per thread* remains constant. The aim is to also keep the execution time constant. We can observe that the execution time increases by 60% from 1 to 96 threads. Generally speaking, this can come from two factors :

- synchronizations between threads;

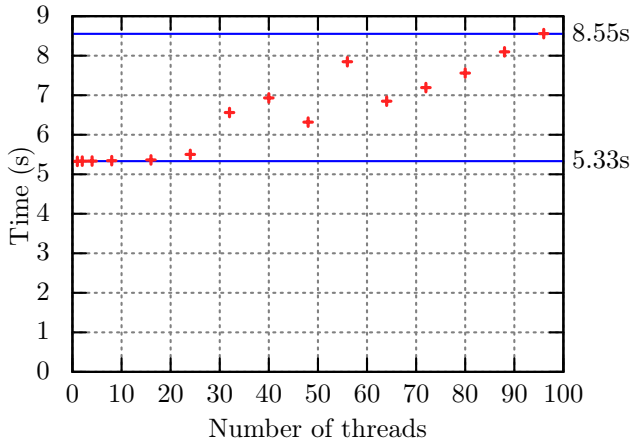

Figure 4: Weak scalability of our prallel program. The experiment was carried out with $1,000,000 \times N_{\text{threads}}$ simulations and 1,000 runs.

- frequency decrease for keeping a tolerable CPU temperature.

Our program barely contains thread synchronization. The only such region is the reduction for computing the sum of all the explored branches. Profiling showed that this region represents less than 1% of execution time. We can conclude that the cause must be a decrease of the CPU frequency.

No CPU can run at maximum frequency all of its cores. If only a few of them are doing intensive calculations, they can get the maximum frequency. However, if too many cores do so, the frequency has to decrease. If it did not, the CPU would melt.

Figure 5 shows how our program scales with more threads on a constant problem size. Figure 4 already showed that the speedup is not perfect. At that point, using all the 96 core of the Graviton 4 processor, the experiment ran within 2.89s.

Landmark 3

By using the ARMPL RNG, an optimal random number buffer size, and all the cores of the processor, we reached the performance of 10Gbranch/s. Compared to the base program, the speedup is 435.

2.5 Saving Computations

The base code had the structure represented by algorithm 1.

In the deepest loop, the one over $N_{\text{simulations}}$, we can observe that every value but Z is constant. Therefore, we can define alternative constant values not to perform the same calculations mutiple times.

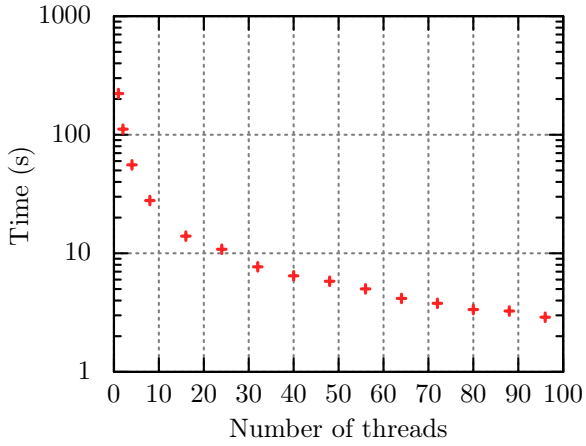

Figure 5: Strong scalability of our parallel program. The experiment was carried out using 30,000,000 simulations and 1,000 runs.

$$\alpha \leftarrow (r - q - \frac{\sigma^2}{2})T \tag{1}$$

$$\beta \leftarrow \sigma\sqrt{T} \tag{2}$$

Then, profiling our new version of the code showed that the computation of the exponential was starting to get expensive, reaching up to 20% of the total time. Sometimes, the exponential is computed but not actually used. At line 6 of algorithm 1 we see that there is a condition, we actually need the value of the exponential if, and only if $\max(S_T - K, 0) = S_T - K$. Let's try to re-write this condition.

$$\begin{aligned}
\max(S_T - K, 0) = S_T - K &\iff S_T - K > 0 && (3)\\
&\iff S_T > K && (4)\\
&\iff S_0 e^{(r-q-\frac{\sigma^2}{2})T+\sigma\sqrt{T}Z} > K && (5)\\
&\iff S_0 e^{\alpha+\beta Z} > K \quad \text{(using (1) and (2))} && (6)\\
&\iff e^{\alpha+\beta Z} > \frac{K}{S_0} \quad (S_0 > 0) && (7)\\
&\iff Z > \frac{\ln(\frac{K}{S_0}) - \alpha}{\beta} \quad (\beta > 0) && (8)
\end{aligned}$$

The benefit of the form (8) of the condition is that the whole right-hand side of the inequality is *constant*. At this point, we can know just by comparing our random number Z to a constant value whether we have to compute the exponential or not. However, performing this test may not be interesting.

Algorithm 1 Structure of the basic code.
N_{runs}, $N_{\text{simulations}}$, K, q, r, S_0, T, σ, are constant values defined at the beginning of the program.

Procedure black_scholes
$S_{\text{payoffs}} \leftarrow 0$
for j **from** 1 to $N_{\text{simulations}}$ **do**
    $Z \leftarrow$ random_gauss()
    $S_T \leftarrow S_0 \exp\left((r - q - \frac{\sigma^2}{2})T + \sigma\sqrt{T}Z\right)$
    payoff $\leftarrow \max(S_T - K, 0)$
    $S_{\text{payoffs}} \leftarrow S_{\text{payoffs}} +$ payoff
end for
return $\frac{\exp(-rT)S_{\text{payoffs}}}{N_{\text{simulations}}}$

Procedure main
$S \leftarrow 0$
for i **from** 1 to N_{runs} **do**
    $S \leftarrow S +$ black_scholes()
end for
return $\frac{S}{N_{\text{runs}}}$

As a matter of fact, performing the test saves time if the expensive computation is avoided, but if it is not, then we spent extra CPU cycles to perform a worthless test. Three different situations can occur.

1. In the vast majority of cases, computing the exponential is not needed. This is the case where performing the test is useful: the times we avoid the expensive computation compensates for the overhead on the iterations where we cannot avoid it.

2. The opposite, the few cases where the exponential is avoided do not make the check overhead worth.

3. Balanced, depending on the RNG, we can either save or waste a bit of time.

In order to know which situation we were in, we simply implemented the test and counted the number of iterations where the computation could have been avoided. This empirical method showed that the proportion of exponentials that have to be computed never goes beyond 35%. We are in case number 1, which makes the test useful.

Adding this test into the loop actually resulted in worse performances, increasing the time spent in the exponential to 30% and adding on top 6% spent on the branch statement, as seen in the Linaro Map report in figure 6.

This decrease in performance can be firstly explained by the branching that can cause mispredict in the CPU speculative execution and is also preventing

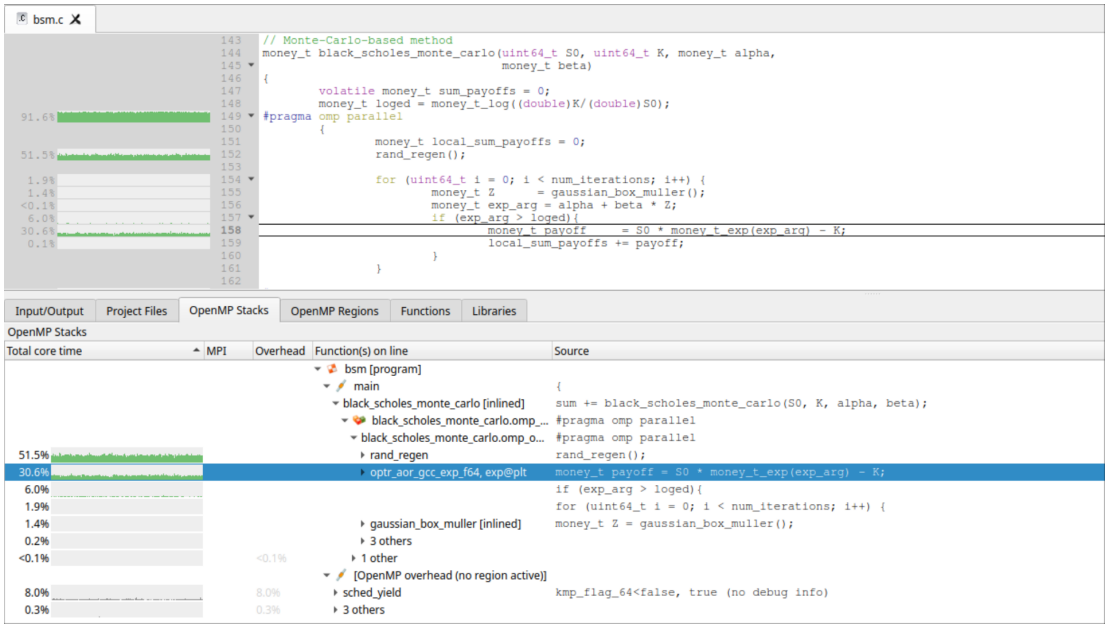

Figure 6: Linaro Map profiling report of the code with branching to only compute the needed exponentials.

the compiler from vectorizing the loop. As seen in the profiling report, the instruction called for the exponential is `exp_f64` which is not a vectorized instruction. How can we implement this idea without branching and keeping the vectorization?

2.6 Vectorization

Well, it turns out that both of these issues can be resolved by vectorizing the code by hand using intrinsics. We are used to Intel AVX intrinsics thanks to academic projects that we carried out this year. However, no one in the group had ever worked with ARM SVE vectorization, so it was the perfect opportunity to learn it using some ressources[5].

The idea is simple, fill a vectorized register with the generated random numbers Z, compare it, strictly greater, to a vectorized register filled with the constant term seen in (8). This gives us a mask that we use to compact the original register, keeping only the elements where the mask is true and puts them to the left side of the register, then we can count the number of true elements in the mask and use this result to store this many elements from the vector to a buffer in memory. This process is visualized by figure 7.

This way no branching is done as the booleans are stored and used as masks.

[5] https://developer.arm.com/architectures/instruction-sets/intrinsics/

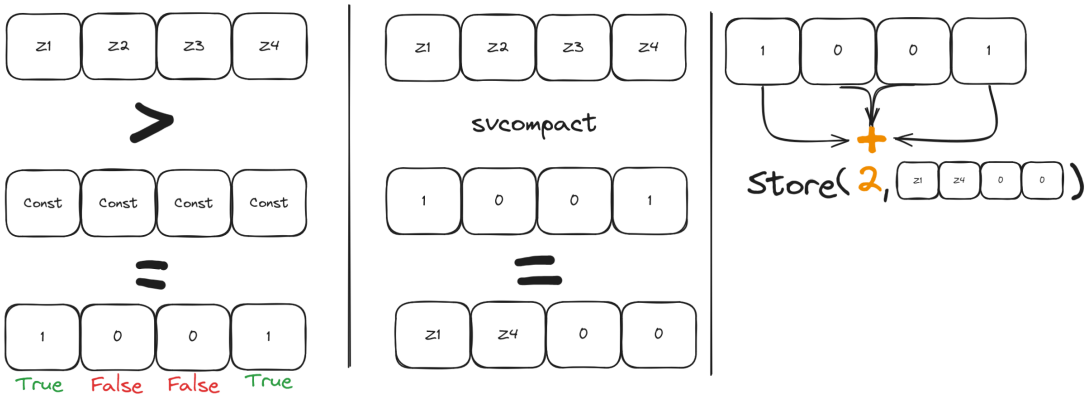

Figure 7: Diagram of the vectorization process. Register sizes are for demonstration only and are not equivalent to the actual sizes.

After the loop over $N_{\text{simulations}}$, we have a buffer filled with random number that needs to be put through the exponential function to compute S_T. This is done in a secondary loop also with efficient vectorized instructions such as Fused-Multiply Add (FMA) where possible.

Landmark 4

Using vectorization and saving computation we obtained even better performances, nearly 50% faster, unfortunately values obtained by this method weren't as precise, most likely due to a bug in the code that we didn't have time to find. Given the promising results, fixing it should be a priority.

2.7 Comparison of Graviton 3 and Graviton 4

CPU	Precision	Value	Time (s)	Performance (Gbranch/s)	Correct Decimals
Graviton 3	double	5.136955	1438	6.95	5
Graviton 4	double	5.136955	909	11.0	5

Table 2: Comparison of performance of Graviton 3 and 4. Experiment carried out with 10,000,000 simulations, 1,000,000 runs, and the maximum number of threads: 96 for Graviton 4 and 64 for Graviton 3.

Table 2 shows the gap in performance between the two processors. Graviton 4 is 58% faster than Graviton 3. This difference mainly comes from the fact Graviton 4 has 33% more cores.

2.8 Our Final Results

$N_{\text{simulations}}$	N_{runs}	Value	Time (s)
100,000	1,000,000	N/A	N/A
1,000,000	1,000,000	5.136957	131
10,000,000	1,000,000	5.136955	909
100,000,000	1,000,000	5.136956	12057

Table 3: Benchmark of our final version.

Table 3 shows the results that were asked for in the header of the source file we got. Sadly, we did not run the experiment for the first set of parameters.

From the third to the fourth row of table 3, performance drops from 11.0Gbranch/s to 8.29branch/s. Since the execution lasted for 3 hours and 20 minutes, the cause may be that a member of the group executed a program on the node wile it was computing.

Landmark 5

According to the third row of table 3. Our final perfomance using ARMPL RNG, optimal random buffer size, 96 threads is 11.0Gbranch/s. The speedup compared to the base version is 478.

3 Code Aster

3.1 Compilation

The compilation of Code Aster consists of three main steps :

1. Install a few basic dependencies;
2. Compile and install the prerequisites of Code Aster;
3. Compile and install Code Aster.

The source code for steps 2 and 3 were available on the Code Aster's Gitlab page and thus we can not choose the versions. However, we were able to choose the versions to use for the basic dependencies of step 1.

3.1.1 Main Issues

We will first introduce the main issues that eventually lead us to the final installation of Code Aster. This section will explain the specificities over our choices on packages versions and why our modifications on the Code Aster code may be different from other groups.

We began working on Code Aster trying to use the most up to date basic packages, which are cmake, python, lapack and boost. However, it lead to some issues and difficulties. First, the cmake 3.31.4 version we use is raising many *OLD_POLICIES* warnings, so we would recomand switching to the 3.20 version that fits better with the packages we are installing. Second, we began the project using boost version 1.87.0, but as the *mfront* dependency of Code Aster had troubles finding liboost_python shared library when compiling with armclang, we moved to an older version of the package (version 1.75.0), which worked well with it.

Furthermore, our first attempt at compiling the prerequisites was using GCC 14.2 and OpenMPI 5. Unfortunately, even though we managed to compile and install Code Aster, we had an issue at execution time. In fact, when executing in distributed with more than one processus, the libaster core raised an **APPELMPI_95** error. After talking with Conrad Hillairet, we narrowed down multiple possibilities that could trigger this error. Finally, we discovered that we were tricked by a preinstalled modulefile called **OpenMPI5**. We though this modulefile was loading the OpenMPI5 environment installed for us, but in reality this modulefile was loading an OpenMPI5 library available in the system. We had to repeat the installation with the right installation of OpenMPI to get a working installation of Code Aster. Moreover, as we were trying multiple fixes at the same time, our successful installation was compiled with GCC 13 and OpenMPI 4. Also, we suspect that our installation would compile successully with GCC 14.2 and OpenMPI 5 even though we did not have the time to try it.

Moreover, as one of our dependencies is python, we tried using a local installation of it, so we could use the most up to date version, with optimization flags on. However, we noticed that even though we correctly set up our environment variables and used **hash -r** to clear the cache, the **which python** command was still referring to the system installation of python. Therefore, we chose a radical but successful way to fix the issue by moving **/usr/bin/python** to **/usr/bin/python.bak**, thus forcing bash to use our local python binary over the system one.

Then, as we did not put a correct architecture in the **make ROOT=... ARCH=...** command for the prerequisites at first, we may have add some modifications to the source code as we were compiling a sequential version of the Code Aster.

Finally, to sum up the aim of this section, you may have understood that we have tried versions for libraries. Therefore you have to be warned that we used an incremental approach to compile the Code Aster prerequisites. Thus, our GCC 13 installation is using the modifications we made for GCC 14. Moreover, the version compiled with armclang is based on the modifications made for the GCC compilation. We have also kept some modifications made while we had the wrong architecture put in the make command. This means that we may have some modification that are unnecessary and that would come from older work on the prerequisites.

3.1.2 Compiling and Installing Code Aster with GCC

As mentioned briefly above, Code Aster prerequisites have some dependencies over cmake, python, boost and lapack. The version of these libraries we are using in the given installation script are :

- cmake 3.31.4
- Python 3.9
- lapack 3.12.1
- boost 1.75.0

The reason we are compiling these libraries locally is mainly because the yum package manager is not able to install up to date versions of these libraries. Also, as we compile these libraries from source, we can choose some compilation flags. This way, we compiled Python with the following flags : `--with-pymalloc --with-valgrind --enable-optimizations --enable-shared`. As Code Aster code is using Python as an interface with the user, we used the three first flags to facilitate the performances of Code Aster, and make the library easier to debug. We add the last flag preventively as we know that some prerequisites may need the python's shared library.

We worked with the 20240327 version of Code Aster prerequisites as we are aiming at compiling the latest stable version of Code Aster. The compilation of these dependencies represents the main difficulty of Code Aster compilation. Consequently, we will address and explain each issues and its fix individually.

libmetis We noticed that the given installation script of the prerequisites was skipping the installation of the libmetis. This is an issue as it is needed later by the mumps prerequisite. Therefore, we chose to install the libmetis version 5.1.1 alongside its Gklib dependency.

mfront During the installation of mfront, the g++ compiler could not find the declaration of the `std::find` function. Therefore, we had to add the `#include <algorithm>` line in one of the files to make it compile. Also, when we switched from boost 1.87.0 to 1.75.0 with armclang, a new issue was raised. This issued mentioned a enum variable that could be assigned with a negative id, which was not allowed. Therefore, we replaced the assigned id by the minimum between itself and 0 to remove the error.

ScaLAPACK During the compilation of ScaLAPACK, the calls to `IGAMX2D` function raised a `rank mismatch error`. We found that this error was a common issue coming from the use of a newer version of GCC. Thus, we added a `-fallow-argument-mismatch` to fix it. Moreover, the ScaLAPACK compilation raised a `implicit declaration` error on the `BI_imvcopy` in a C file. We tried to ignore this error with a `-Wno-error=implicit-function-declaration` flag, and checked wether function was found at link time. As no error was raised during linking time, we concluded that this fix was enough.

mumps The mumps dependency showed the same issue as in ScaLAPACK, with the fortran compiler rising a `rank mismatch error`, so we chose to fix it the same way by adding a `-fallow-argument-mismatch`. Also, mumps could not link to functions from the metis and zlib library. Consequently, we added the required flags to make rectify this error.

miss3d This library raised a `rank mismatch error` too, that we fixed with the **-fallow-argument-mismatch** once again. Moreover, it was using a `-mcmodel=medium` flag that was not recognized by the gfortran compiler. This flag is handling the size of the binary and code functions, and is usually set to small by default. In fact, in a x86 architecture, a call to a function is translated by a *jump* instruction, that is using an offset to reach the desired function code. However, if the gap in memory between the *jump* instruction and text section of the function is too long, then the jump can not be done with a mcmodel=small behavior because the offset is too large to fit in 32 bits. However, the Graviton processors are using an Aarch architecture, which default behavior is to add intermediate trampolines for function calls. Therefore, the only mcmodel implemented is the

small one, which is the one by default. Thus, we removed this flag from the Makefile, as it is not needed on our architecture.

MEDCoupling First, this library raised an error as it was trying to convert a negative **char** to a positive one. In fact, on x86 architectures, a `char` is signed by default, whereas it is unsigned on ARM architectures. To fix it, we add a `-fsigned-char` and a `-Wno-narrowing` flag. Secondly, another error was raised as a conversion from **`_object*`** to `const tagPyArrayObject_fields*`. As this issue only appeared when we switch from Python 3.8 to Python 3.9, and as a pointer to an `object` can be equivalent to a pointer to an `ArrayObject`, we assumed it was correct to add an explicit cast in the code to remove the error, which worked. Thirdly, we had an issue with the size of the `med_int` typedef, defined by the med library. In fact, a med_int is a long integer by default, but the MEDCoupling library was trying to cast it to a integer, which was not correct. By searching deeper, we noticed that we could change the size of a med_int during the configuration of the med library by adding `--with-med_int=int`. Therefore, the use of this flag fixed the issue ... until the compilation of Code Aster's core, where it is assumed that med_int is a long integer, which raised other casting errors. Thus, we came back on the size of a med_int, by using the default `--with-med_int=long` behavior again. Finally, we noticed an issue in the source code of MEDCoupling, as the `-DMED_INT_IS_LONG=1` was not set. This way, we added this flag by passing it as a CFlag to the CMakeLists, which fixed the issue.

Code Aster core Once all the libraries are installed, the last step is to install Code Aster's core, available on gitlab. To do so, we run :

```
./configure --prefix="$HOME/install/code_aster" \
            --python="$HOME/install/python-3.9/bin/python3.9" \
            --enable-mpi \
            --enable-openmp

./waf install --jobs=$(nproc)
```

The configure command is calling the appropriate `waf configure` command to configure the Code Aster project. As we are using a local installation of python, we need to give the path to our python binary. Also, with the 2 last flags, we make sure that Code Aster will enable all its multi-threaded and distributed algorithms. Finally, we can compile Code Aster's core on multiple cores using the `jobs` option.

3.1.3 Compiling and Installing Code Aster with Armclang

Now that we have successfully compiled Code Aster with GCC 13, we would like to compare the previous installation performances with one compiled with a compiler optimized for ARM, like armclang. Therefore, we will need to compile every dependency with it. The only exception will be the libboost, which does

not allow it, and would need additional work. Therefore, the libboost will be compiled with GCC.

For the prerequisites, we used our previous modifications as a baseline for armclang. The first modification we have made to this baseline was to remove all the `-fallow-argument-mismatch` we added for gfortran, as the issue is not raised by armflang. The second one was to correct a typography error in the mgis library, where a call to `this->result.has_value` has been written as `this->result_has_value`. We have no doubt this is a typography error, because the surrounding code is using the correct function call.

Finally, we need to compile Code Aster's core. However, we lacked time for this part and did not manage to complete it. The issue we got was the configuration script not able to check our fortran compiler. However, we believe that the installation code we are giving you (see Section 3.1.4) will be able to compile Code Aster's core with the arm compiler. In fact, we think that we only forgot an environment variable for the Fortran MPI wrapper.

3.1.4 Installation Script

Alongside this report, you will find an archive that manages the installation of Code Aster. To run our compilation script of Code Aster, we highly recommend you to take a look at the **README.md** in the archive. The archive contains one folder managing the compilation with gcc 13 and OpenMPI 4, and another folder for the compilation using armclang and OpenMPI 5.

As mentioned earlier, we can not affirm that the compilation of Code Aster with armclang will be successful, but you should be at least able to compile the prerequisites. Moreover, you might see in the patches we apply to the prerequisites of ScaLAPACK, that we add some link flag to some MPI libraries. We are nearly sure that these links should be removed, but as we did not have the time to test it, we let them here.

Finally, as told in Section 3.1.1, your `/usr/bin/{python,python3}` will be moved to `/usr/bin/{python,python3}.bak` by our installation script.

3.2 Benchmark

The code was compiled using the **-O3** optimization flag to enhance its performance. The most recent Boost C++ library was employed to maximize efficiency.

Benchmarks were conducted on both Graviton 3 and Graviton 4 nodes by varying three parameters. Tests included two solvers, with *Code Aster* being compiled under different configurations. These solvers were evaluated using varying node counts, and levels of refinement.

3.2.1 The REFINE

The benchmark problem, provided by EDF, simulated the displacement of a cube. The complexity of the computation was controlled using the REFINE environment variable, which dictated the precision of the simulation. Computational effort, measured in floating-point operations per second (FLOPS), increased exponentially with the refinement size.

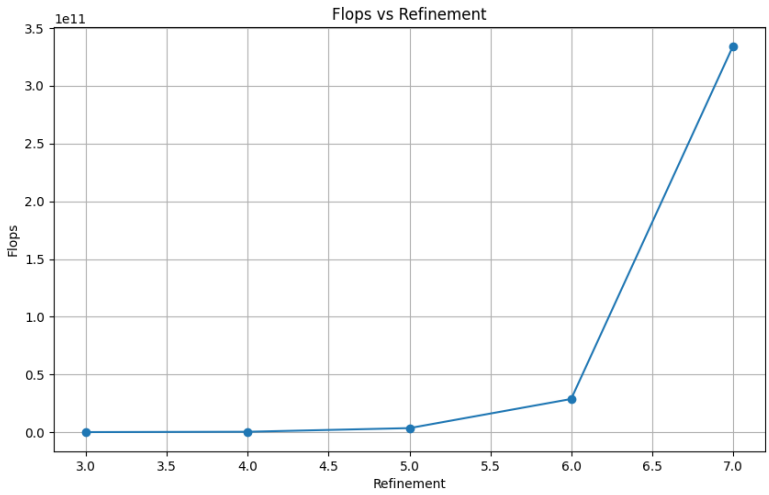

Figure 8: FLOPS vs Refinement Level based on PETSc performance reports.

The FLOPS presented in the previous graph approximate the computational workload based on performance reports from the PETSc library. These reports offered insights into the library's workload distribution.

To prevent overly long execution times, the tests were run with the –time-limit=600 (in seconds) option. The maximum refinement level used was 7. Both computational demands and data size increased exponentially with higher refinement levels. Even with all cores fully utilized on a single node, simulations with a refinement size of 8 failed. These failures caused nodes to enter the "DOWN" state, rendering them unusable until manual intervention from the UCit support.

The same if the problem size was too small and the number of cores too high, executions also failed. This occurred because the program could not adequately partition the workload. For this reason, refinement sizes smaller than 4 were not tested.

3.2.2 Graphical Analysis of PETSc

Figure 1: Total Time vs. Number of Processors on Refinement 7 for Graviton 3 and Graviton 4

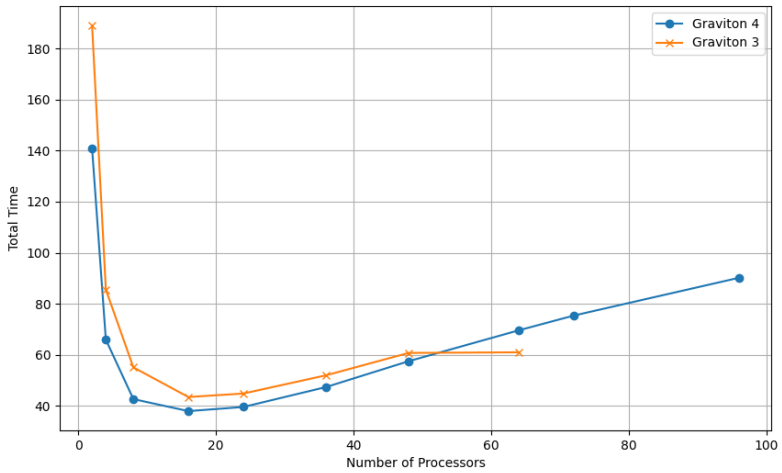

Figure 9: Total Time vs Number of Processors on Refinement 7 for Graviton 3 and Graviton 4 using PETSc.

The performance of the PETSc library scales well initially as the number of processors increases, but performance deteriorates once the processor count exceeds 24. This behavior is notable despite the total computational workload remaining constant. The diminishing returns beyond 24 processors indicate a bottleneck likely caused by factors external to raw computation.

Figure 2: CPU + System Time vs. Total Time on Graviton 4

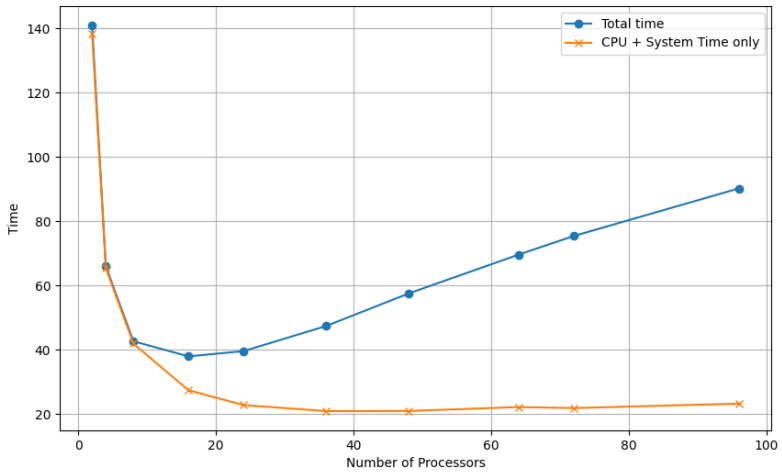

Figure 10: CPU + System Time compared to Total Time on Graviton 4 using PETSc.

This observation is reinforced by Figure 2, where the CPU + System Time remains constant beyond 24 processors, suggesting that the amount of compute remains unchanged. However, the total time increases significantly, implying that the additional time is spent on other operations.

Figure 3: Volume of Communications vs. Number of Processors on Refinement 7 for Graviton 4

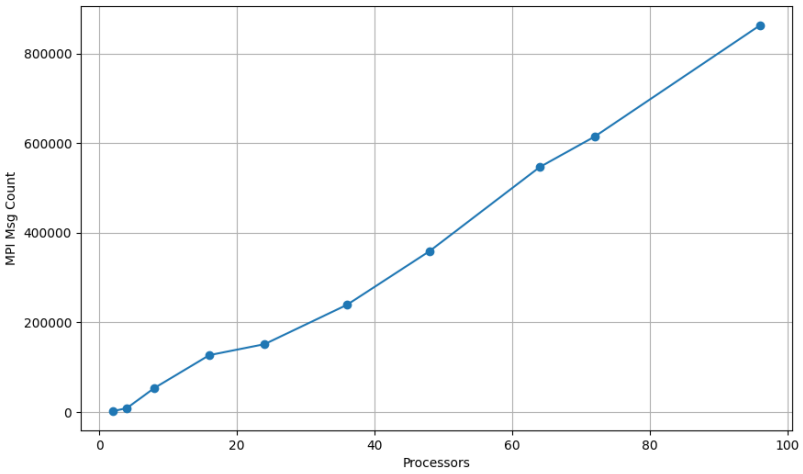

Figure 11: Volume of Communications vs Number of Processors on Refinement 7 for Graviton 4 using PETSc.

The sharp increase in communication volume, as shown in Figure 3, explains why performance scales poorly beyond 24 processors. As additional processors are introduced, the volume of inter-processor communication grows disproportionately. The parallel efficiency degrades because the time spent coordinating processors with communications outweighs the gains achieved through parallel computation.

3.2.3 Analysis of MUMPS

The information provided during executions with the MUMPS library was less detailed, with execution time being the primary metric. Additionally, a warning appeared during testing, indicating that an acceleration option specified for the MUMPS solver was unsupported by the current version. The solver replaced the unsupported option with a fallback labeled "FR." While this substitution enabled computations to proceed, the warning highlighted the potential for reduced performance.

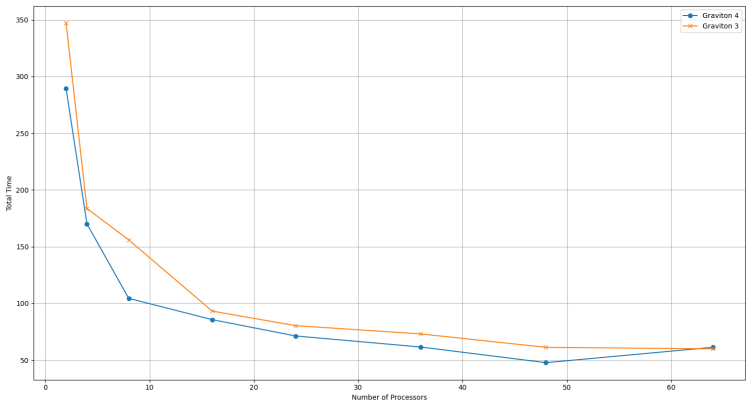

Figure 12: Total Time vs Number of Processors on Refinement 6 for Graviton 3 and Graviton 4 using MUMPS

The highest refinement level tested with MUMPS was 6, as tests with higher refinement levels frequently timed out. Nevertheless, refinement level 6 demonstrated good scalability across both Graviton 3 and Graviton 4 nodes.

In comparison, PETSc outperformed MUMPS when using fewer processors, consistent with EDF's test results. However, some data points for Graviton 4 are missing from the graph because the timeout value was insufficient for these tests.

3.3 Validation

Our program runs the cube example without any issues, both in distributed and sequential modes, across all refinement sizes. Aside from the previously mentioned problems, where the issue is either too large a problem with insufficient resources or too small a problem that can't be properly partitioned.

However, 49 out of 2335 tests fail. We did not have time to explore this issue further and chose to focus on benchmarking instead.

3.4 Further work on Code aster

Due to time constraints, we were unable to fully optimize the code or explore additional performance-enhancing strategies, as a significant portion of our efforts was dedicated to getting the compiling of a functional implementation.

One area of improvement would have been to compile the code using more optimized flags and ARM-specific compilers and libraries. Additionally, addressing PETSc's scalability challenges could have been possible by implementing intra-node parallelism. Combining OpenMP for shared-memory parallelism with MPI

for inter-node communication could have mitigated the communication overhead that currently limits scalability. Furthermore, exploring alternative libraries beyond MUMPS and PETSc might have offered new avenues for optimization and improved performance.