Parallelizing applications using Hedgehog

Report from engineer student

3^{rd} year internship

Option 2: Software engineering and information systems

Presented by : **Rémi CHASSAGNOL**

Company's tutor: Walid KEYROUZ
Academical tutor: David HILL

internship length: 5 months
Oral: 5/10/2024

Campus des Cézeaux. 1 rue de la Chébarde. TSA 60125. 63178 Aubière CEDEX

Acknowledgements

J'aimerais remercier Loïc Yon pour sa confiance et pour avoir rendu ce stage au NIST possible. J'adresse aussi mes remerciements à mon tuteur académique, et professeur, David Hill pour ses précieux conseils et sa flexibilité. Je tiens aussi a remercier grandement Alexandre Bardakoff pour ses conseils techniques, et également sur la vie aux États-Unis. Je le remercie aussi pour tout le travail qu'il a réalisé au NIST avant mon arrivée. Merci aussi aux professeurs d'anglais et de communication, Susan Arbon Leahy et Murielle Mouzat, dont les cours, d'une grande qualité, m'ont beaucoup aidé durant ce stage.

I would like to thank Timothy Blattner for his guidance during the project, as well as Walid Keyrouz his flexibility and his advice.

Finally, I express my thanks to the contributors of the LaTeX project, that was used to write this report, as well as the developers hunspell and ChatGPT that helped to proofread the text.

Abstract

This report describes the research performed at NIST, for optimizing a fire simulation. The objective was to improve the computing performance of FDS (Fire Dynamic Simulator), on single computing node, using a C++ library called Hedgehog. This library represents parallel algorithms as data-flow graphs. The report also describes the improvements that have been made to a serialization library, with the objective of utilizing it to transfer data on the network on the cluster version of Hedgehog.

FDS is composed, amongst other algorithms, over Cholesky decomposition. A version using Hedgehog was implemented to demonstrate the library's capabilities. We eventually managed to have a slightly better performance than the OpenBLAS implementation. Some parts of the fire simulation have also been optimized. Eventually, we came to the conclusion that parallelizing FDS by hand was too difficult, and we decided to implement a tool that could automate a part of the process. This tool is briefly introduced in this report, however, it is not implemented yet.

For the serialization library, some new features have been added. However, despite the fact that some parts of the library have been optimized, the current performance are still far below the best serialization libraries available. The library still proposes a unique design, and more optimizations will be made in the future.

Keywords: **C++**, **HPC**, **Hedgehog**, **FDS**, **simulation**, **serialization**.

Résumé

Ce rapport décrit le travail, réalisé au NIST, sur un projet consistant en l'optimisation d'une simulation de feu. L'objectif était d'améliorer les performance de calcul de FDS, sur un unique nœud de calcul, en utilisant une bibliothèque nommée Hedgehog. Cette bibliothèque utilise des data-flow graphs pour modéliser des algorithmes parallèles. Ce rapport décrit aussi les améliorations qui ont été apportées à une bibliothèque de sérialisation, avec pour objectif de l'utiliser pour transférer des données sur la version cluster de Hedgehog (version utilisée sur un cluster de calcul).

FDS utilise la décomposition de Cholesky. Une version utilisant Hedgehog a été implémentée pour démontrer les capacités de la bibliothèque. Nous avons finalement réussi à obtenir de meilleures performance que l'implémentation de OpenBLAS. Des parties de la simulation ont aussi été optimisées. Finalement, nous avons conclu que paralléliser FDS à la main été trop difficile. Nous avons donc décidé d'implémenter un outil capable d'automatiser en partie cette tâche. Cet outil est brièvement décrit dans ce rapport, cependant, il n'est pas encore implémenté.

Quant à la bibliothèque de sérialisation, de nouvelles fonctionnalités ont été ajoutées. Cependant, bien que certaines parties de la bibliothèque aient été optimisées, les performances actuelles sont encore bien en dessous de celles des meilleures bibliothèques du marché. La bibliothèque propose tout de même des choix de conception unique, et plus d'optimisations seront apportées dans le futur.

Mots-clés: **C++**, **HPC**, **Hedgehog**, **FDS**, **simulation**, **serialisation**.

Contents

Tableau des illustrations

List of Tables

List of Figures

List of listings

Introduction

In this report, we will describe a project conducted at the National Institute of Standards and Technology, as part of the third year internship at the engineering school of ISIMA.

The purpose of the project was to optimize a fire simulation program, called Fire Dynamic Simulator, using a C++ library named Hedgehog, developed at the NIST. The simulation code is currently able to scale on multiple nodes, however, the performance within a single node could be improved. Our goal is to use Hedgehog and some HPC (High Performance Computing) techniques to accelerate the computation of the simulation within a node.

Another objective will be to participate in the development of Hedgehog. At present, Hedgehog does not support running on multiple nodes. However, some researchers at the university of Utah are currently working on a port of Hedgehog to clusters. To achieve their goal, they will need a serialization library in order to transfer the data on the network. Here, we plan to use a serialization library originally developed as part of an ISIMA project. This library will need to be optimized in order to be usable in HPC applications.

The report begins with an overview of Hedgehog and its features. We then describe the implementation of the Cholesky decomposition (used in FDS) using the library. This will serve as a first example on how Hedgehog can be used. In the third part, we will explain FDS and some of the work that has been realised on the simulation's code. Following that, we will briefly introduce a new project aimed at creating a tool that will assist porting FDS to Hedgehog. We then detail the modifications that have been brought to the serialization library before concluding the report.

Part I
Context of the project

1 The NIST

The National Institute of Standards and Technology (NIST) is an agency of the United States Department of Commerce (DOC) [12]. The National Bureau of Standards (NBS), also known as the National Metrological Institute (NMI), was founded in 1901, and was renamed NIST in 1988. The NIST's role is to promote innovation and industrial competitiveness of the United States. Its mission is composed of three main core functions: developing measurement technology and techniques, creating a maintaining standards, and supporting advancement in technology. The researche conducted at the institute impacts several areas such as Health, Energy, Manufacturing, Cybersecurity, Artificial Intelligence, High Performance Computing, and more. The NIST facilities are distributed across two main campus, Gaithersburg, Maryland and Boulder, Colorado.

NIST is composed of five laboratories:

- The *Physical Measurement Laboratory* (PML) is responsible for finding innovative precise measurements methods in physics, chemistry, and engineering.
- The *Material Measurement Laboratory* (MML) perform research on materials, nanotechnologies and biomaterials.
- The *Information Technology Laboratory* (ITL) develops IT standards and works mainly on Artificial Intelligence, Cybersecurity and High Performance Calculus.
- The *Communications Technology Laboratory* (CTL) works on core network technologies, wireless systems or public safety communications.
- The *Engineering Laboratory* (EL) works on technology for engineered systems in order to enhance economic security and improve quality of life.
- Finally, the *Center for Neutron Research* (NCNR) provides world-class neutron measurement capabilities to meet the needs of researchers from industry, academia, and government.

Each laboratory is subdivided into divisions and the divisions are split into groups. This internship has been realized in the *Information Systems* group directed by Walid Keyrouz. This group is a part of the *Software and Systems* divisions of the Information Technology Laboratory located on the Gaithersburg campus. The division is directed by Ram D. Sriram.

2 Objectives

The objective of the internship is to study the usage of Hedgehog and data-flow graphs in the context of HPC applications. One of the project's goals is to use Hedgehog to accelerate a simulation application called FDS.

Hedgehog is a HPC library written in C++ and developed at the NIST. It uses data-flow graphs to express algorithms composed of tasks that can be run asynchronously. This library will be described in Section 1.

FDS is a fire simulation also developed at the NIST. The simulator reads a configuration file, solves the fluid dynamics equations and outputs the results into files that can be analyzed directly or used with other programs. Currently, FDS is written in Fortran (standard 2018); it is parallelized with MPI and OpenMP. The objective is not to study the mathematical model behind the simulation, but rather to study how the code is organized and how it processes the data, to be able to design a Hedgehog graph that will orchestrate the simulation and improve its multicore performance.

Another goal of the internship is to contribute to the development of Hedgehog. Currently, the library does not support running on multiple nodes on clusters. The current cluster version of Hedgehog uses MPI to handle communication between the nodes. However, this requires serializing the data before the communication. To solve this problem, it has been proposed to use a serialization library written at ISIMA during the third-year project. The objective will be to optimize the serialization time and extend the features of the library in order to be able to use it in Hedgehog.

3 Problem analysis

The first step is to understand how to use Hedgehog. To do that, the easiest way to do this is by following the tutorials and creating initial programs to explore the various features of the library.

The next step will be to study FDS's source code and isolate the different parts that will be parallelized using Hedgehog. FDS is written in Fortran and some part of the code are already parallelized with MPI or OpenMP. Originally, the objective was not to rewrite the entire application in C++. We aimed to split the simulation code into functions that could be used in C++. Only a few critical algorithms should have been entirely rewritten in C++ in order to optimize the performance. All the preliminary tests were conducted with this approach in mind, however, we will see that we eventually concluded that this approach was not feasible because of the technical limitations of the compilers[1]. Furthermore, when the project has been started, our goal was to port FDS to Hedgehog manually. Once again, after some first tests, we finally decided to automate the process. To do that, our goal is to create a refactoring tool that will perform AST (Abstract Syntax Tree) transformations and that will generate the code of the Hedgehog graph. This report will briefly describe the specifications of this tool, but it will not detail its development since this project has just begun. It is good to mention that this could not have been predicted without a prior knowledge of the simulation's code, and the first tests that have been made have played an important role in the definition of the specifications of the tool.

Concerning the serialization library, the primary optimization idea was to use binary serialization instead of a human-readable format. Indeed, the library originally used a JSON like format, however, the binary one is lighter and does not need the usage of a parser. Furthermore, another optimization will be to use strings instead of streams. Indeed, as we will explain, the streams are usually implemented with a linked list. They require to frequently access the RAM of the computer whereas strings are arrays that can be easily loaded in the CPU cache.

Some features will also be added to the library in order to make it more powerful. The first version like the support for static and dynamic arrays. Furthermore, some data structure can be difficult to serialize because of pointers. A workaround to this issue is to give the possibility to the user to execute code during the serialization. Finally, the old version used some macros that were generating a lot of code and that was very difficult to maintain. These macros need be removed in order for the library to be more usable.

[1]These limitations are imposed by the standard and not the compilers themselves.

Part II
Realization and design

1 Hedgehog

Hedgehog is a template library designed to create parallel implementations using data-flow graphs. It is the direct successor of HMBE (HTGS Model-based engine), which also utilized data-flow graphs [1]. Hedgehog was developed as part of Alexandre Bardakoff's PhD thesis at NIST [2].

In this section, we will present some of the features of the library and how we can use it to express an algorithm under the form of a graph.

1.1 The data-flow graphs

As mentioned in the introduction of this section, algorithms are expressed as data-flow graphs. A data-flow graph is a graphical representation where edges denote data streams and vertices represent tasks as explained by [3] and [2]. An example of a data-flow graph is shown in Figure 1.

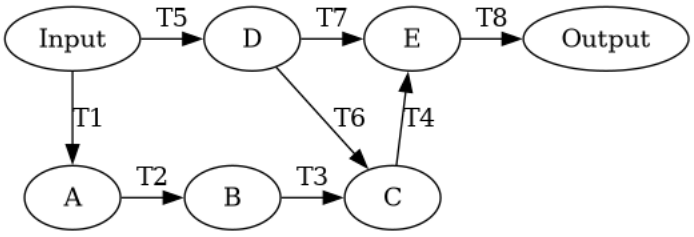

Figure 1: Data-flow graph example

In each task, the input data is consumed and transformed into new data that serves as input to other tasks. For instance, in Figure 1, there are two inputs of types `T1` and `T5`. The data of type `T1` is input into Task `A` that produces new data of type `T2` that is transmitted to Task `B`. At the end of the graph, Task `E` generates data of type `T8` that is the output of the graph.

In Hedgehog, the data are transmitted between nodes using shared pointers to ensure efficiency even, with large data types.

1.2 The nodes

To design an algorithm using Hedgehog, we create a graph in which the data will flow through various nodes. These nodes can be of different types, which we describe in this section.

1.2.1 Tasks

The first type of nodes are the tasks. These nodes are made to be duplicated and run in multiple threads. As all other nodes, a task can have multiple input and output types.

On Listing 1, we can see how to create a task using Hedgehog. To do this, we define a class that inherits from `hh::AbstractTask` and we override the `execute` and `copy` functions. The `execute` functions are called when the task receives a data of a certain type and the `copy` function is used to duplicate the task to run on multiple threads.

The tasks have at least three template parameters. The first one is the number of input types (in this example, we have two input types) and it is followed by the list of the input types (T3 and T6) and the list of the output types (T4).

```
 class C: public hh::AbstractTask<2, T6, T3, T4> {
 public:
   SubLineTask(size_t nbThreads):
       hh::AbstractTask<2, T6, T3, T4>("C", nbThreads) {}

   void execute(std::shared_ptr<T3> data) override {
     // compute with T3 data
     return std::make_shared<T4>(/* ... */)
   }

   void execute(std::shared_ptr<T6> data) override {
     // compute with T6 data
     return std::make_shared<T4>(/* ... */)
   }

   std::shared_ptr<hh::AbstractTask<2, T6, T3, T4>>
   copy() override {
       return std::make_shared<C>(this->numberThreads());
   }
};
```

Listing 1: Hedgehog: Task

1.2.2 States

States are special nodes used only for synchronisation and cannot be duplicated. They are thread-safe and control the data-flow in the graph. It is important to be careful when creating a state as it locks the program using a mutex. This means that only simple operations have to be made in the states in order not to slow the program execution.

Creating a state is very similar to creating a task. A state inherits from `hh::AbstractState` and overrides the `execute` function. However, states do not have a `copy` function, as they should not be parallelized.

A state cannot be used directly in a graph and must be managed by a state manager, which is described in Section 1.2.3.

1.2.3 State Managers

A state manager is a node that owns a state. It handles locking the state when data is received. Multiple state managers can own the same state in a graph, which allows reuse of the state at different stages of the computation. Hedgehog provides a default implementation for the state manager, but users can create custom ones to handle cycles.

One of the key features of Hedgehog is the fact that it allows the creation of cycles in the graphs. This is particularly useful for repeating operations (loops in computation). However, it is not possible to automatically detect when a cycle can be broken at runtime. Users can implement their own state managers and override the `canTerminate` function to control when a state should stop.

Listing 2 provides an example of a state manager for a type `MyState`. Here, the `canTerminate` function is overridden to return whether the state is finished. In this function, the state is locked, and the `isDone` method (defined in the state) is used to know if the state can be finished. The template parameters of the `StateManager` should match the parameters used when creating the state.

```cpp
 class MySM: public hh::StateManager<1, T1, T2, T3, T4> {
 public:
   MySM(std::shared_ptr<MyState> const& state):
       hh::StateManager<1, T1, T2, T3, T4>(state, "My SM") { }

   [[nodiscard]] bool canTerminate() const override {
       this->state()->lock();
       auto ret = std::dynamic_pointer_cast<MyState>(this->state())->isDone();
       this->state()->unlock();
       return ret;
   }
};
```

Listing 2: Hedgehog: state manager

1.2.4 Graphs

Graphs themselves are nodes as well, meaning that a graph can contain other graphs. In a graph, nodes are connected by edges in order for them to be able to communicate.

Listing 22 shows how to create a graph. In this example, some nodes are created (tasks and states), and the edges are defined to connect these nodes. For instance, at Line 12, an edge between `myTask1` and `myTask2` is created, meaning that the task one (the sender) will output data to the task two (the receiver). To create an edge between two nodes, they must share at least one type (if task one outputs a data of type `T1`, the task two should accept this type as input). Like other nodes, graphs have inputs and outputs, which can be specified using the corresponding methods.

```cpp
class MyGraph: public hh::Graph<1, T1, T2> {
  public:
    GaussGraph(size_t matrixHeight, size_t nbThreads):
        hh::Graph<1, T1, T2>("My Graph") {
      auto myTask1 = std::make_shared<MyTask1>(/* nbThreads, ... */);
      auto myTask2 = std::make_shared<MyTask2>(/* nbThreads, ... */);
      auto myState = std::make_shared<MySate>(/* ... */);
      auto myStateManager = std::make_shared<MySM>(myState);

      this->inputs(myTask1);
      this->edges(mayTask1, myTaks2);
      this->edges(mayTask1, myStateManager);
      this->edges(mayTask2, myStateManager);
      this->edges(myStateManager, mayTask2);
      this->outputs(myStateManager);
    }
};
```

Listing 3: Hedgehog: graph

The graph class can be instantiated to create a graph to which data can be pushed as shown in Listing 4. After pushing the data, the function `finishPushingData` is called to start the computation, and `waitForTermination` is used to wait for the end of the computation.

```cpp
MyGraph myGraph;

myGraph.executeGraph();
myGraph.pushData(/* ... */);
myGraph.finishPushingData();
myGraph.waitForTermination();
```

Listing 4: Hedgehog: using a graph

1.3 Profiling

Hedgehog includes a powerful profiling tool that generates a *dot* file. This file can be compiled using *graphviz* [13] to generate an image representing the graph. The generated graph contains detailed information on the execution of the graph such as the execution time, the size of the queues, the number of threads, and more. This information can be used to optimize the graph by balancing the number threads between the tasks or optimizing the tasks that are slower. It can also be used to refactor the algorithm as it is very easy to identify the critical path on the image.

An example of a compiled dot file for a graph performing the Hadamar product on two matrices is shown in Figure 2. This example is presented in the tutorials in the Hedgehog's documentation [14].

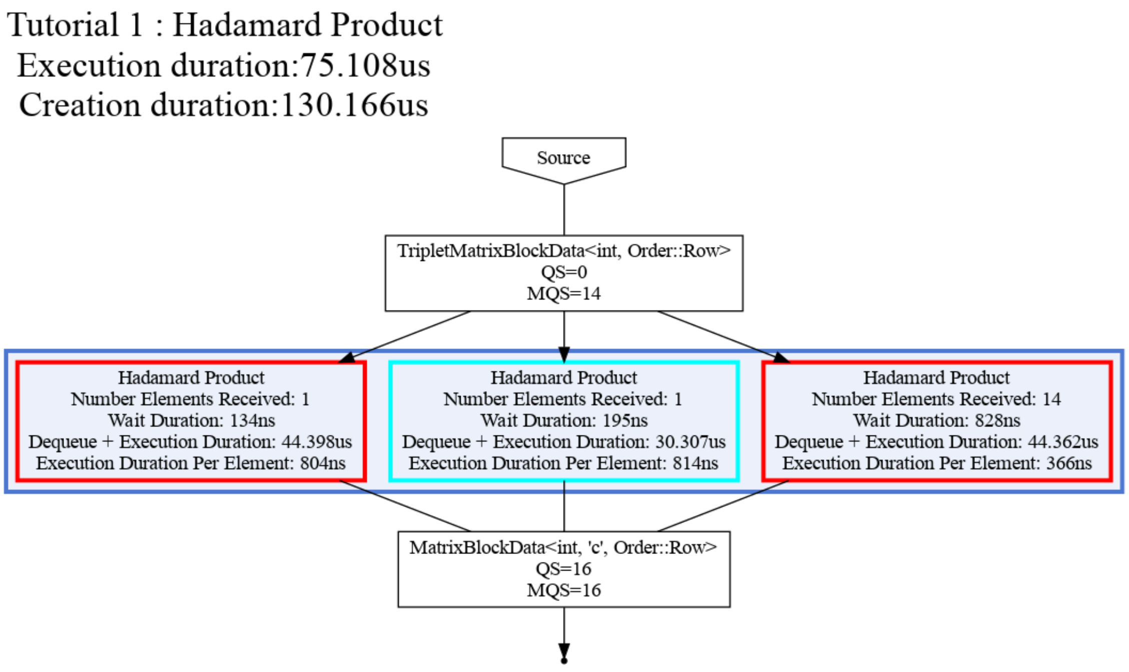
Tutorial 1 : Hadamard Product
Execution duration:75.108us
Creation duration:130.166us
Source
TripletMatrixBlockData<int, Order::Row>
QS=0
MQS=14
Hadamard Product
Number Elements Received: 1
Wait Duration: 134ns
Dequeue + Execution Duration: 44.398us
Execution Duration Per Element: 804ns
Hadamard Product
Number Elements Received: 1
Wait Duration: 195ns
Dequeue + Execution Duration: 30.307us
Execution Duration Per Element: 814ns
Hadamard Product
Number Elements Received: 14
Wait Duration: 828ns
Dequeue + Execution Duration: 44.362us
Execution Duration Per Element: 366ns
MatrixBlockData<int, 'c', Order::Row>
QS=16
MQS=16

Figure 2: Output of the profiling tool (Hadamar Product)

It is important to note that the profiling tool is always active and incurs minimal additional cost to the execution time of the graph [2].

2 Cholesky

The initial task undertaken during the project was the implementation of the Cholesky decomposition. The work that is described in this section was realized to demonstrate the capabilities of Hedgehog, and it was done without any knowledge of how FDS operates. For these reasons, we will describe this work before explaining the principles of the simulator.

FDS's simulation requires solving equations using the Cholesky decomposition algorithm. However, the current implementation in the simulation code is not fast enough. In this section, we will start by explaining the algorithm. Then we will explain the implementation with Hedgehog before presenting the results. For the measurements, our implementation will be compared to the OpenBLAS's implementation as it is one of the fastest currently available.

2.1 The algorithm

The Cholesky algorithm solves systems of linear equations of the form $Ax = y$, where $A \in M_{n,n}(\mathrm{R})$ is symmetric and positive-definite [15]. The principle is to decompose the matrix A into a matrix L such that $A = LL^T$, where $L \in M_{n,n}(\mathrm{R})$ is a lower triangular matrix. The algorithm then solves the two sub-problems: $Lx' = y$ and $L^T x = x'$.

There are several ways to implement this algorithm. For this project, we implemented the block version as it is the most suited for parallel computing. Indeed, one of the most important concepts of HPC is optimizing CPU cache usage. To achieve this, we will decompose our matrix into smaller blocks that can be entirely loaded into the cache, minimizing the need for the CPU to access the RAM.

The version of the algorithm that we will use is described in [16]. This algorithm proceeds in three steps. First, consider the matrix A shown in Figure 3. In this example, D represents a diagonal block, C is a column (composed of multiple blocks) and A' is a sub-matrix.

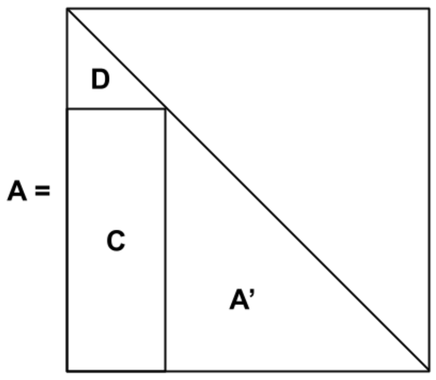

Figure 3: Cholesky decomposition: matrix A

The first step of the algorithm is to process the diagonal element D. To do this, we use the sequential version of the algorithm. It will be fast enough given the size of the block. The second step is to update the elements on the column beneath the diagonal block, by solving the system $C_{\text{new}} = C(D^T)^{-1}$. Once this is done, the algorithm has computed the first column of the result matrix L. The final step consists of updating the rest of the matrix A by performing $A' = A' - CC^T$.

Once we have obtained the matrix L, we can proceed to the solving part to find the solutions of the system. This process can be decomposed in two steps: first, we solve the equation involving the diagonal block, then we update the rest of the matrix and the result vector using the partial solution (we remove the variables from the equation). We repeat these steps for each diagonal block to solve a sub-problem. The solver must be applied twice to solve the two sub-problems $Lx' = y$ and $L^T x = x'$.

2.2 Implementation

The implementation of the algorithm was done in two parts. We began by implementing the decomposition, and once it was fully functional, the solver part was added. This approach was chosen because the decomposition is the most critical part of the algorithm. It was important to compare our version with the OpenBLAS's one before implementing the solver.

The first implementation of the decomposition involved creating a graph that followed the steps described in Section 2.1 similarly to OpenBLAS. This graph is shown in Figure 4.

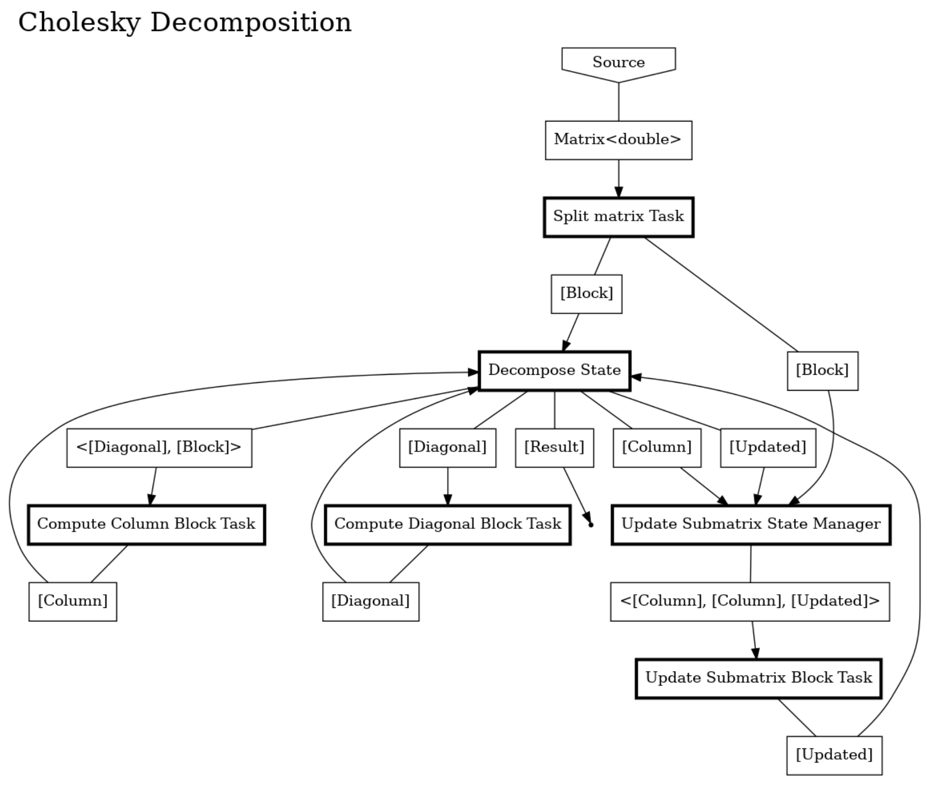

Figure 4: Cholesky decomposition graph

In this diagram, tasks and states are represented by the bold rectangles and the data by the regular rectangles. There are two kinds of data that flow into this graph. The first kind is the `Matrix` that is the input of the graph and the second kind is the matrix blocks represented by the block's identifier enclosed in `'[]'`. The matrix is given to the `Split Task`, which generates the blocks. The blocks have identifiers, which define their types. These identifiers are used to send the right blocks to the appropriate task (or state). For instance, when the `Decompose State` sends a `Diagonal` block, it is transferred to the `Compute Diagonal Block Task`. The execution of this graph can be visualized as follows:

1. Split the matrix into blocks (`Split Matrix Task`).
2. Compute the diagonal element **C** (`Compute Diagonal Block Task`).
3. Compute the elements on the column **D** (`Compute Column Block Task`).

4. Update the sub-matrix **A'** (`Update State / Task`).
5. Restart from 2 with the blocks of **A'** (and send the result blocks).

This first solution was too slow because the three steps were executed sequentially (which is also what is done in OpenBLAS). By doing so, we were loosing a lot of parallelism. For instance, the computation of the diagonal elements could be done with only one thread and all the other threads had to wait. To optimize the algorithm, we aimed to keep the threads as busy as possible, and the solution for this was to rely on early computation by processing the blocks as soon as they are ready. For instance, we can begin solving the next diagonal element before all the blocks of the sub-matrix A' are updated. Similarly, we can handle the column the same manner.

To make the early computation possible, the block's data structure was modified. A `rank` attribute was added to keep track of the modifications on the blocks. Each time a block is updated, its rank is incremented. When the rank of a block is equal to its column number, it is ready to be processed, and it is processed when its rank is superior to its column number. The states use pending lists to store indices of blocks that should be processed, and iterate on these lists to launch the computation of all ready blocks. The graph maintains the same structure, but the operations in the state have been drastically changed.

In this second version, several verifications are required, using the rank in the states. Furthermore, both of the states have a vector of block pointers that are shared between them to make sure that when the *decompose state* changes a rank, the *update state* can be aware of the modification directly, without any notification. This improves the performance but adds complexity in the states.

Eventually, we managed to have good performance with this second version as OpenBLAS does not make any early computation. We will present our results in the section 2.3.

For the solver, we use a second graph that is employed twice to solve the two sub-systems of the matrix. The graph is shown in Figure 5.

Here, we observe that there are two sub-graphs for the solving part. These two graphs solve the two sub-problems that we described earlier. The first solver is directly connected to the decomposition graph, and it receives the decomposed block. Both solver graphs are connected to the split task, and they receive the split vector blocks.

In this graph, we use the same matrix blocks as in the decomposition graph, allowing us to start the solving part directly when the decomposition graph outputs the first result block. The key principle of Hedgehog is to allow the creation of pipelines in which the data flow continuously, so we avoid having unnecessary barriers[2] in the process. Here, we do not have to finish the decomposition entirely to start solving the blocks. This helps to maintain maximal parallelism and ensure the computer's resources are used efficiently. In other words, if we represented the computation time between the logical cores in a Gantt chart, our goal would be to optimize the overlap between tasks and minimize gaps as much as possible (ideally, the cores should never be in a wait state).

[2]Points in the program where we must wait for the completion of task before starting the next one.

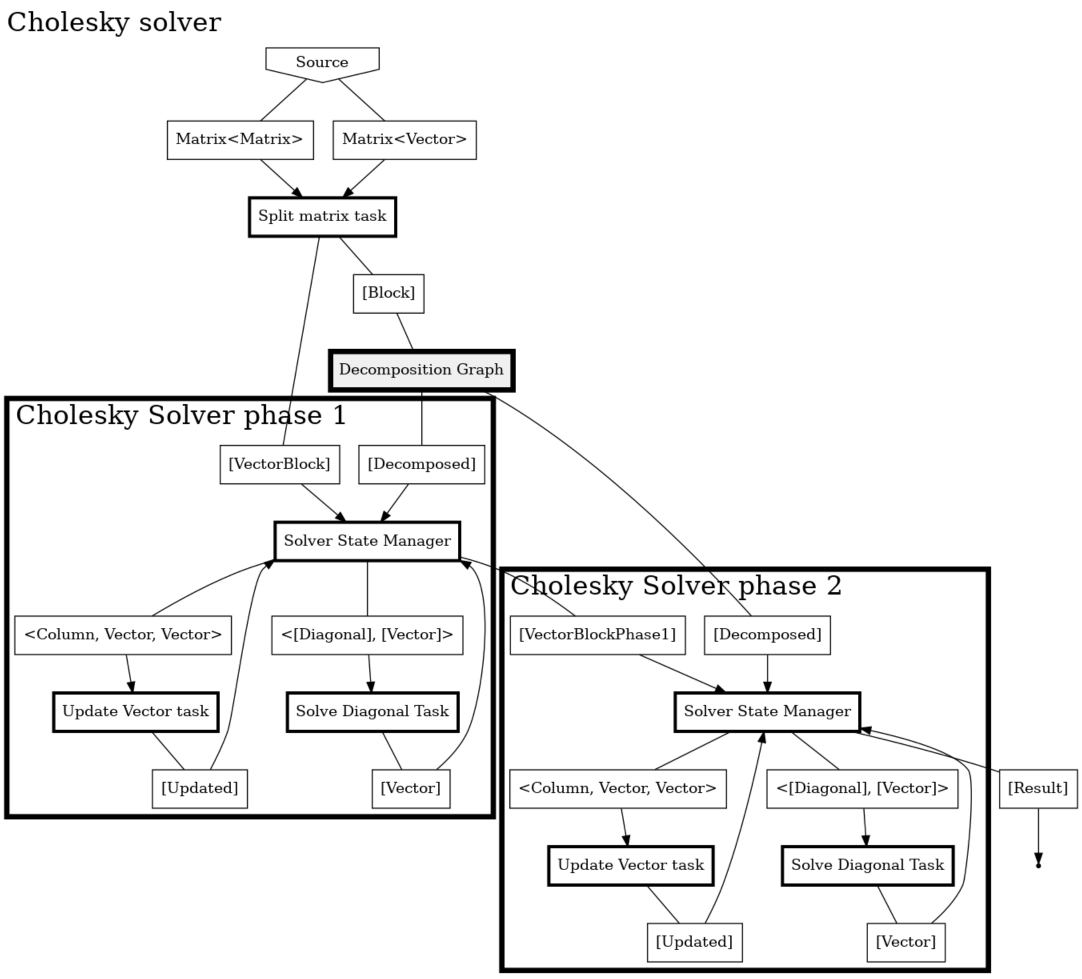

Figure 5: Solver graph

2.3 The results

Now that we have explained the algorithm and its implementation, we will analyse the results of the measurements on the decomposition algorithm using both the OpenBLAS's implementation and the Hedgehog's one. Two sets of measures have been made on a cluster node with 192 CPU cores. The specifications of the cluster are listed in Appendix A. For the first set, the problem size was 40,000 and the program has been run with a maximum of 256 threads. The second set uses a 100,000 matrix and a maximum of 384 threads (utilizing all the cores). For each problem, the measurements include the computation times of the two algorithms for different number of threads (from 1 to the maximum) to see the relative speedup. With Hedgehog, the number of threads was changed only for the most critical task (the update task). It was fixed to the optimal numbers for the other tasks (these numbers were found by doing first tests measures beforehand).

First, let us analyse the measurements made on the 40,000 matrix. Figure 6 shows the graph generated at the end of the program using Hedgehog's profiling tool. It indicates that the bottleneck of the program is indeed the update task as we explained before. We can see as well that a single thread was allocated to the diagonal task and 12 were allocated to the column task. The diagonal task does not require more than one thread since it processes at most one block at a time (this is shown by the maximum queue size `MQS` on the rectangle before the task). For the column task, it appears relatively slow (as indicated by the color). We could allocate more threads to it, but we have to make sure that we keep enough resources for the update task.

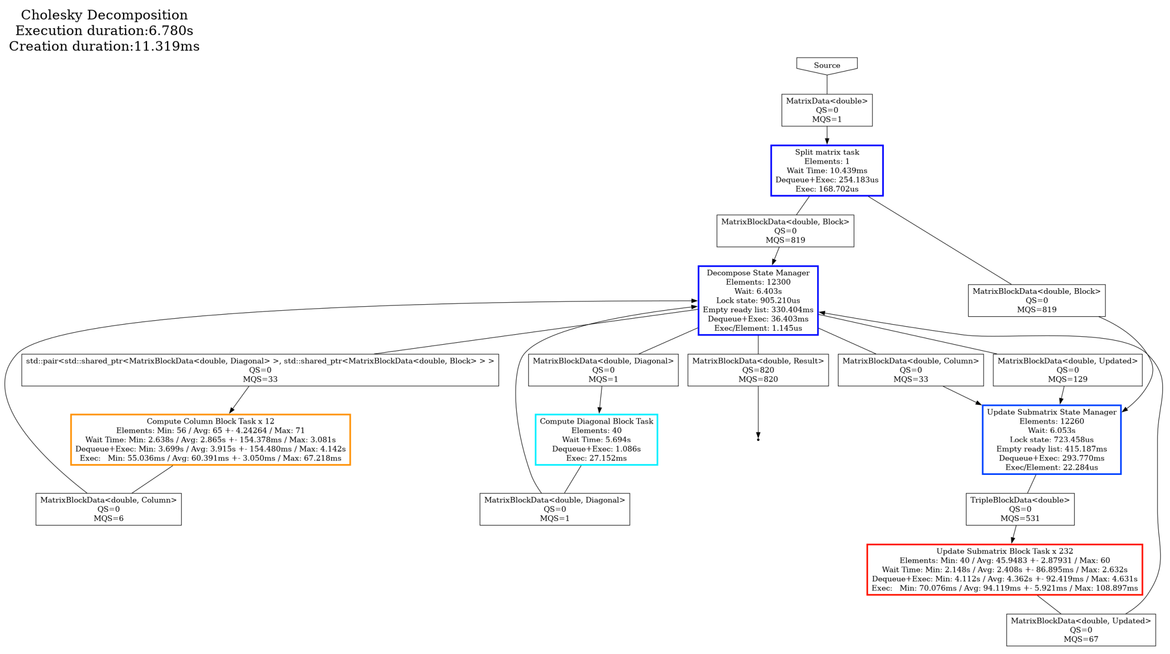
Cholesky Decomposition
Execution duration:6.780s
Creation duration:11.319ms
Source
MatrixData<double> QS=0 MQS=1
Split matrix task Elements: 1 Wait Time: 10.439ms Dequeue+Exec: 254.183us Exec: 168.702us
MatrixBlockData<double, Block> QS=0 MQS=819
MatrixBlockData<double, Block> QS=0 MQS=819
Decompose State Manager Elements: 12300 Wait: 6.403s Lock state: 905.210us Empty ready list: 330.404ms Dequeue+Exec: 36.403ms Exec/Element: 1.145us
std::pair<std::shared_ptr<MatrixBlockData<double, Diagonal> >, std::shared_ptr<MatrixBlockData<double, Block> > > QS=0 MQS=33
MatrixBlockData<double, Diagonal> QS=0 MQS=1
MatrixBlockData<double, Result> QS=820 MQS=820
MatrixBlockData<double, Column> QS=0 MQS=33
MatrixBlockData<double, Updated> QS=0 MQS=129
Compute Column Block Task x 12 Elements: Min: 56 / Avg: 65 +- 4.24264 / Max: 71 Wait Time: Min: 2.638s / Avg: 2.865s +- 154.378ms / Max: 3.081s Dequeue+Exec: Min: 3.699s / Avg: 3.915s +- 154.480ms / Max: 4.142s Exec: Min: 55.036ms / Avg: 60.391ms +- 3.050ms / Max: 67.218ms
Compute Diagonal Block Task Elements: 40 Wait Time: 5.694s Dequeue+Exec: 1.086s Exec: 27.152ms
Update Submatrix State Manager Elements: 12260 Wait: 6.053s Lock state: 723.458us Empty ready list: 415.187ms Dequeue+Exec: 293.770ms Exec/Element: 22.284us
MatrixBlockData<double, Column> QS=0 MQS=6
MatrixBlockData<double, Diagonal> QS=0 MQS=1
TripleBlockData<double> QS=0 MQS=531
Update Submatrix Block Task x 232 Elements: Min: 40 / Avg: 45.9483 +- 2.87931 / Max: 60 Wait Time: Min: 2.148s / Avg: 2.408s +- 86.895ms / Max: 2.632s Dequeue+Exec: Min: 4.112s / Avg: 4.362s +- 92.419ms / Max: 4.631s Exec: Min: 70.076ms / Avg: 94.119ms +- 5.921ms / Max: 108.897ms
MatrixBlockData<double, Updated> QS=0 MQS=67

Figure 6: Hedgehog graph for the 40,000 matrix

In Figures 7 and 8 we see the computation times of the two implementations and the speedup of the Hedgehog's version compared to the OpenBLAS's one for the 40,000 matrix. Here, we observe an interesting behavior for the OpenBLAS program. As the number threads increases, the Hedgehog program continues to scale, while the OpenBLAS's one becomes unstable around 100 threads. A possible explanation is that OpenBLAS is optimized for non-hyper-threaded configurations. For these measurements, we allocated 256 threads on the cluster which means that we could have used 1 thread per core till more than 128 threads were allocated to the program. This number is quite to the point where the instability begins to appear on the OpenBLAS's measurements.

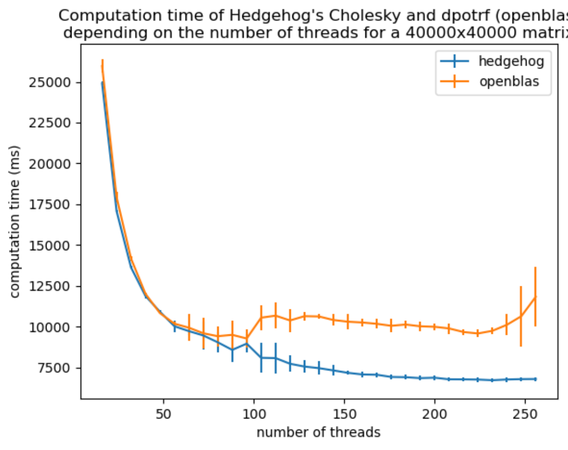

Figure 7: Computation times for a 40,000 matrix

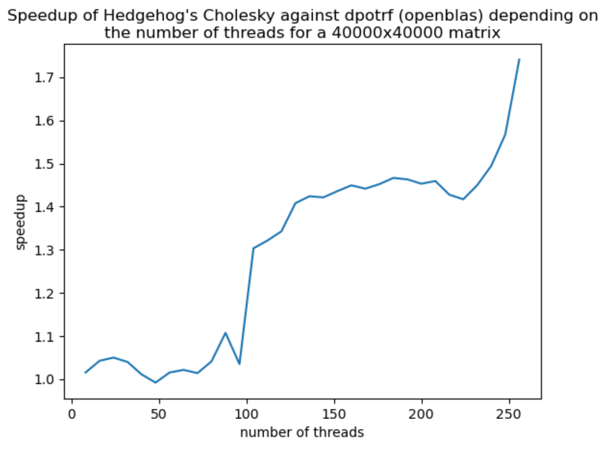

Figure 8: Speedups for a 40,000 matrix (hedgehog / OpenBLAS)

Figure 9 shows the relative speedup for each program ($\frac{time_{one_thread}}{time_{n_thread}}$). As we can see, neither program achieves a significant speedup. This can be attributed to the fact that the current problem is too small, limiting resource utilisation. For this reason, we conducted more measurements with a bigger problem.

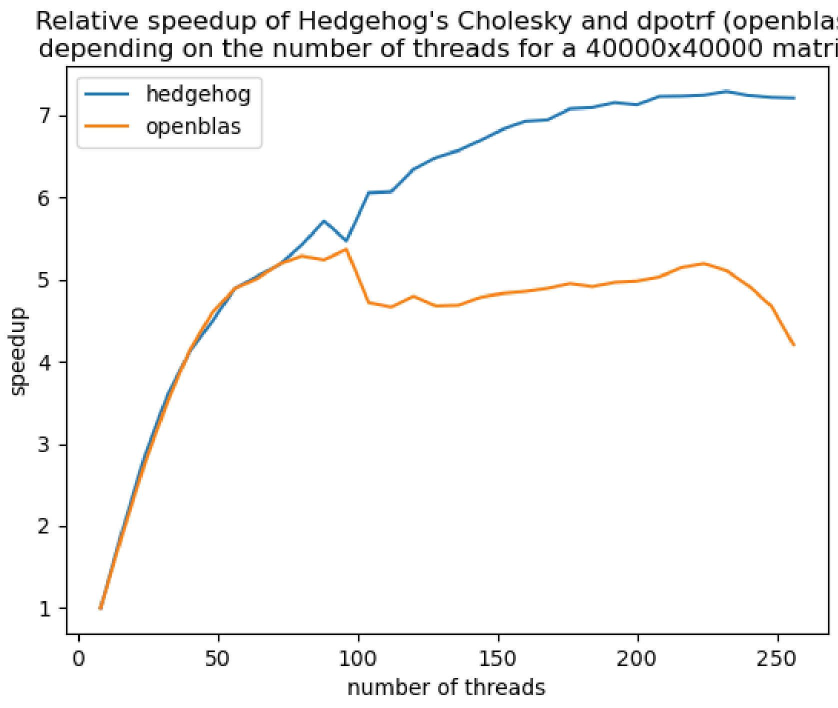

Figure 9: Relative speedups for a 40,000 matrix

The second set of measurements used a $100,000 \times 100,000$ matrix, and utilized up to 384 threads. Figures 10 and 11 show the computation times of the two implementations as well as the speedup of Hedgehog against OpenBLAS as observed previously. These two plots show similar results than the previous ones.

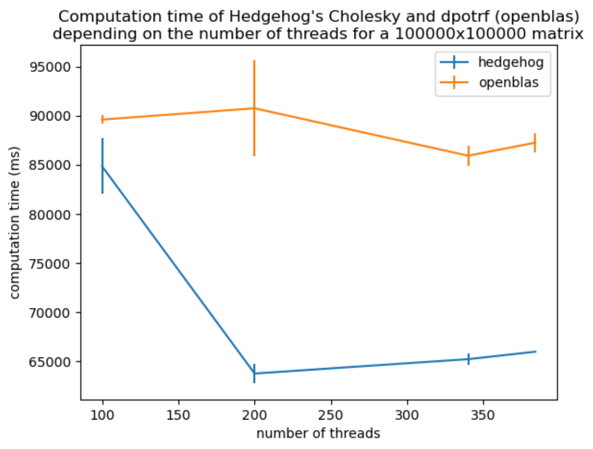

Figure 10: Computation times for a 100,000 matrix

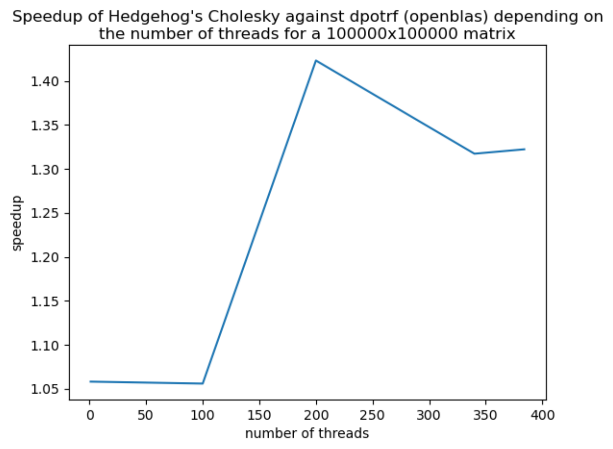

Figure 11: Speedups for a 100,000 matrix (hedgehog / OpenBLAS)

Figure 12 shows that we have a way better relative speedup for the two algorithms. This means that the resources are better exploited with a bigger problem.

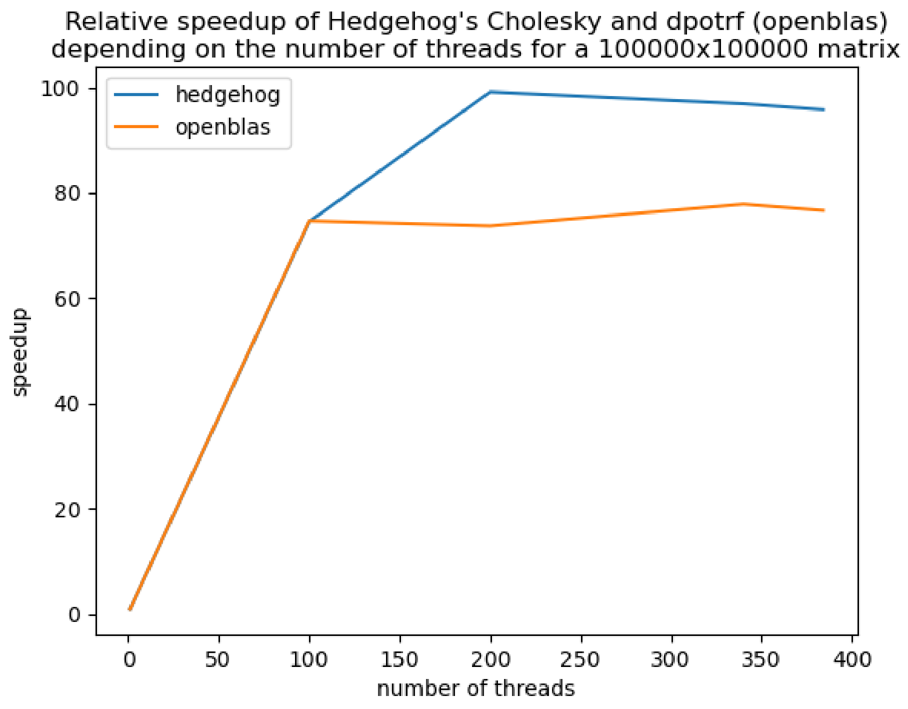

Figure 12: Relative speedups for a 100,000 matrix

Ultimately, we achieved to have better performance than OpenBLAS by incorporating more early computation. Additionally, the program remains readable and easy to understand. This demonstrates the effectiveness of the data-flow graph approach in parallel computing, as well as the capabilities of the Hedgehog library.

3 FDS

In this section, we will explain what FDS is, and we will describe some of the work that has been done on this tool.

3.1 The Fire Dynamics Simulator

"FDS is a computational fluid dynamics (CFD) model of fire-driven fluid flow. The FDS software solves numerically a form of the Navier-Stokes nist-equations appropriate for low-speed, thermally-driven flow, with an emphasis on smoke and heat transport from fires" [17]. The first version of the simulator was published in February 2000. It is used for solving practical problems in fire protection engineering and to study fundamental fire dynamics and combustion [18].

The simulator program reads the configuration from a text file that describes the parameters, computes the solution of the equations and writes the results to different files. The format of the configuration file is described in the user's guide [4]. The output files of the simulator can either be analyzed directly or they can be used as input for other programs. Indeed, FDS can be paired with two other programs. Evac is the evacuation simulation module for FDS; it is used to simulate the movement of people in evacuation situations [5]. Smokeview is a visualization program that displays the output of FDS. Figure 13 shows an example of the Smokeview's output. The usage of this tool is described in [6].

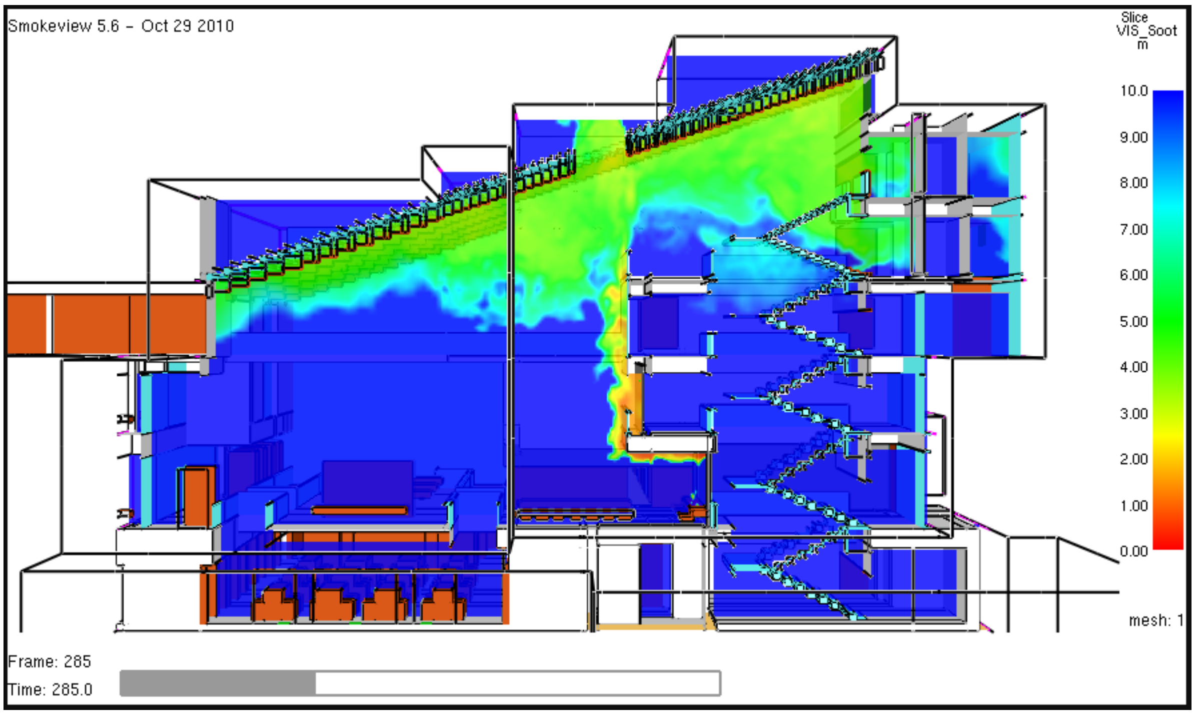

Figure 13: Smokeview example

During the project, neither Evac nor Smokeview were used. However, it is still useful to mention them to provide some context.

FDS uses a large-eddy simulation (LES) model [7], which is a mathematical model for turbulence used in computational fluid dynamics [19]. This is an alternative to direct numerical simulation (DNS) when the computational costs become too high. The idea is to ignore the small elements by utilizing a low-pass filter on the equations to simplify the computation as explained in [19] and [8]. The small elements can still have an effect on the solution. To solve this problem, FDS simulation is divided in two parts as described in the technical reference [7]. First the predictor part estimates the terms at the next time step. At the end of the predictor, the simulator verifies the stability conditions. If the stability conditions are satisfied, the predictor proceeds to the next time step. Otherwise, the corrector adjusts the terms before the simulation continues to the next time step. Both of these steps will be represented as sub-graphs in the Hedgehog version of the simulator. The FDS's simulation has many other characteristics that will not be detailed in this report. However, all the mathematical tools and models used by the simulator are detailed by [7].

The simulation is written entirely in Fortran (standard 2018). The code is already parallelized using MPI and OpenMP. MPI is used to parallelize the computation of the meshes [3] on multiple nodes on a cluster, and OpenMP is used to parallelize many of the loops that are present in the program. Interestingly, FDS does not use any other dependencies such as *BLAS* or *LAPACK* that we used in the Cholesky implementation. Furthermore, the simulator is not yet capable of performing GPU computation, however, the developers are interested in using GPUs in the Hedgehog implementation.

Now that we have described what FDS is, let us see what has been done to optimize the simulation with Hedgehog.

3.2 3D loops

One of the bottlenecks of the simulation is the computation of the velocity. This part of the program consists of three 3D loops that iterates on the tensors of each mesh, and can be used several times during a tick of the simulation. Since it is a critical part of the program, the developers of FDS have created a test program, separated from the full simulation, that is used to measure and try various optimizations. This test program has been very useful in the effort of optimizing the simulation with Hedgehog since it was simpler compared to the simulator itself. In this section, we will describe the test program developed by the FDS's developers. Then we will explain how the computation of the velocity has been optimized using block decomposition. Finally, we will discuss the limitations of the method that has been used.

3.2.1 The test program

As mentioned in the introduction of the section, the test program includes only the part of the simulation responsible for the computation of the velocity. The program consists in three 3D loops parallelized with OpenMP. These loops are independent, so they are in a parallel section and there are no barriers between the loops (they run in parallel). Listing 5 shows the loops with the OpenMP pragmas (note that all the nested loops are not represented here).

The test program is divided in two parts. In the first part, all the variables are initialized. We will not explain the initialization since it is not present in the simulator. This initialization is just used to have values on which we can perform computation. The second part of the program is the computation of the velocity. We measure the computation time for this section. Finally, the program calculates the mean for each direction (`FVX`, `FVY` and `FVZ`) in order to have simple values that can be easily verified. The computation time of the mean is not taken into account when we make the measure on the original test program, unlike in the Hedgehog one. We will explain why the Hedgehog program measures the computation of the mean as well in Section 3.2.2. Finally, like in the FDS's code, most of the variables are global, which makes difficult to analyze the dependencies between the tasks.

[3]As explained in the user's guide, a mesh is a portion of the 3D space. Normally, they are used for modeling complex geometrical structures that do not fit in one parallelogram (corridors for instance). However, they can also be used to perform a block decomposition on the matrices.

```fortran
! Compute x-direction flux term FVX

!$OMP PARALLEL PRIVATE(...)
!$OMP DO SCHEDULE(STATIC) PRIVATE(WOMY, VOMZ, TXXP, TXXM, DTXXDX, DTXYDY, DTXZDZ)
DO K=1,KBAR
    ! ...
ENDDO
!$OMP END DO NOWAIT

! Compute y-direction flux term FVY
!$OMP DO SCHEDULE(STATIC) PRIVATE(WOMX, UOMZ, TYYP, TYYM, DTXYDX, DTYYDY, DTYZDZ)
DO K=1,KBAR
    ! ...
ENDDO
!$OMP END DO NOWAIT

! Compute z-direction flux term FVZ
!$OMP DO SCHEDULE(STATIC) PRIVATE(UOMY, VOMX, TZZP, TZZM, DTXZDX, DTYZDY, DTZZDZ)
DO K=0,KBAR
    ! ...
ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Listing 5: 3D loops source code

3.2.2 The method

To introduce Hedgehog in this first part of the simulation, the goal was to keep the entire computation in Fortran. The objective was to be efficient and avoid rewriting the code in C++. Furthermore, all the variables have been kept in Fortran as well. Here, Hedgehog serves as an orchestrator, responsible for allocating and managing the threads in which the Fortran subroutines are called when it is required. All the OpenMP pragmas have been removed since we do not want to perform fine grain parallelism, we want the threads to be entirely managed by Hedgehog.

As mentioned in the introduction of this section, to optimize computation time, we have used block decomposition on the 3D arrays of the simulation. In order to do this, the Fortran code was reorganized in subroutines that are parametrized with the boundaries of the blocks. The declarations of these new subroutines are shown in Listing 6. Here we can see as well that we use the ISO C bindings to be able to call the subroutines from C++. The original loops have been moved in the new subroutines, and modified to iterate over the blocks instead of the whole matrix.

```fortran
SUBROUTINE COMPUTE_X_FLUX(IBEGIN, JBEGIN, KBEGIN, IEND, JEND, KEND) BIND(C,
↪  NAME = "computeXFlux")
SUBROUTINE COMPUTE_Y_FLUX(IBEGIN, JBEGIN, KBEGIN, IEND, JEND, KEND) BIND(C,
↪  NAME = "computeYFlux")
SUBROUTINE COMPUTE_Z_FLUX(IBEGIN, JBEGIN, KBEGIN, IEND, JEND, KEND) BIND(C,
↪  NAME = "computeZFlux")
```

Listing 6: Loop subroutines signature

Listing 7 illustrates how the subroutines are used in the Hedgehog tasks. Here, we can see that the tasks take a block index as input. As explained earlier, the graph only manages blocks indices. Each block has the start index and the end index on each dimension. In order to send the right blocks to the right tasks, the block type includes an identifier as template parameter. This identifier indicates whether the block should be used to compute the x, y or z direction. In the case shown bellow, we want to compute the terms on the x axis, so the identifier of the block is `X`. The same block is transmitted to the `Y` and `Z` tasks using the corresponding identifiers.

```cpp
  void execute(std::shared_ptr<BlockIdx<X>> block) override {
    size_t i = block->i == 1 ? 0 : block->i;
    computeXFlux(&i, &block->j, &block->k,
                 &block->iEnd,
                 &block->jEnd,
                 &block->kEnd);
    this->addResult(block);
  }
```

Listing 7: Hedgehog compute task for the 3D Loops program.

Figure 14 shows that the Hedgehog graph has three steps:

- Initialization: creation of the block indices (has previously mentioned, Hedgehog manages only indices and not the actual data).
- Computation of the velocity: this part holds the loops that compute the velocity for the x, y and z axis. There is one task per axis, and these tasks call the Fortran subroutines with the blocks boundaries. As shown in the graph, all those three tasks are executed in parallel.
- Computation of the mean.

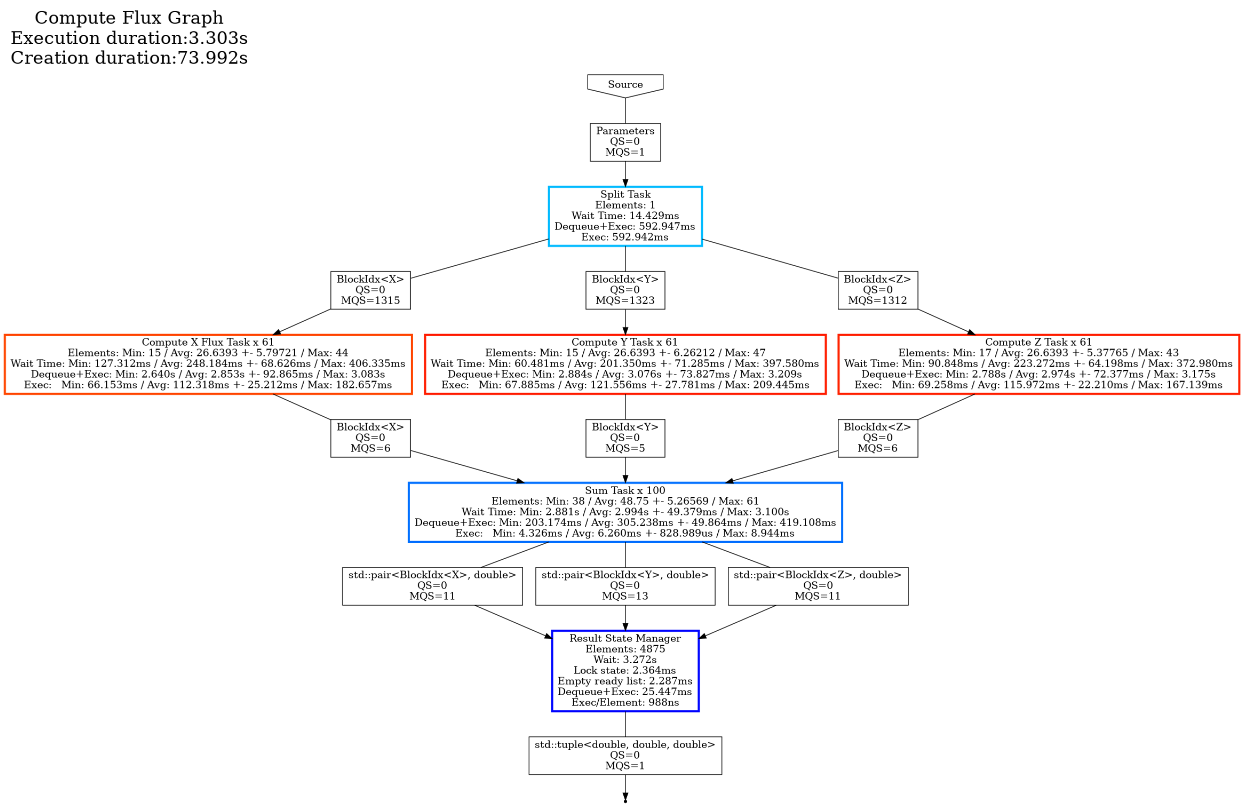
Compute Flux Graph
Execution duration:3.303s
Creation duration:73.992s
Source
Parameters
QS=0
MQS=1
Split Task
Elements: 1
Wait Time: 14.429ms
Dequeue+Exec: 592.947ms
Exec: 592.942ms
BlockIdx<X>
QS=0
MQS=1315
BlockIdx<Y>
QS=0
MQS=1323
BlockIdx<Z>
QS=0
MQS=1312
Compute X Flux Task x 61
Elements: Min: 15 / Avg: 26.6393 +- 5.79721 / Max: 44
Wait Time: Min: 127.312ms / Avg: 248.184ms +- 68.626ms / Max: 406.335ms
Dequeue+Exec: Min: 2.640s / Avg: 2.853s +- 92.865ms / Max: 3.083s
Exec: Min: 66.153ms / Avg: 112.318ms +- 25.212ms / Max: 182.657ms
Compute Y Task x 61
Elements: Min: 15 / Avg: 26.6393 +- 6.26212 / Max: 47
Wait Time: Min: 60.481ms / Avg: 201.350ms +- 71.285ms / Max: 397.580ms
Dequeue+Exec: Min: 2.884s / Avg: 3.076s +- 73.827ms / Max: 3.209s
Exec: Min: 67.885ms / Avg: 121.556ms +- 27.781ms / Max: 209.445ms
Compute Z Task x 61
Elements: Min: 17 / Avg: 26.6393 +- 5.37765 / Max: 43
Wait Time: Min: 90.848ms / Avg: 223.272ms +- 64.198ms / Max: 372.980ms
Dequeue+Exec: Min: 2.788s / Avg: 2.974s +- 72.377ms / Max: 3.175s
Exec: Min: 69.258ms / Avg: 115.972ms +- 22.210ms / Max: 167.139ms
BlockIdx<X>
QS=0
MQS=6
BlockIdx<Y>
QS=0
MQS=5
BlockIdx<Z>
QS=0
MQS=6
Sum Task x 100
Elements: Min: 38 / Avg: 48.75 +- 5.26569 / Max: 61
Wait Time: Min: 2.881s / Avg: 2.994s +- 49.379ms / Max: 3.100s
Dequeue+Exec: Min: 203.174ms / Avg: 305.238ms +- 49.864ms / Max: 419.108ms
Exec: Min: 4.326ms / Avg: 6.260ms +- 828.989us / Max: 8.944ms
std::pair<BlockIdx<X>, double>
QS=0
MQS=11
std::pair<BlockIdx<Y>, double>
QS=0
MQS=13
std::pair<BlockIdx<Z>, double>
QS=0
MQS=11
Result State Manager
Elements: 4875
Wait: 3.272s
Lock state: 2.364ms
Empty ready list: 2.287ms
Dequeue+Exec: 25.447ms
Exec/Element: 988ns
std::tuple<double, double, double>
QS=0
MQS=1

Figure 14: 3D loops graph

It would have been possible to use a single task to perform all the computations for the three axis. However, using three separated tasks means that we also have three queues of data. If only one tasks was used, all the blocks for each axis would be in the same queue. To optimize the dequeue times, we use multiple tasks running in parallel, with each task having its own queue.

Finally, the Hedgehog program computes the mean to demonstrate the pipelining principle of the library. The blocks outputted by the compute tasks are directly accumulated in the sum task. We will see in the next section that even if the Hedgehog program performs this additional operation, it is still faster than the FDS one.

3.2.3 The results

Now that we have explained how the graph has been implemented, we can describe the results. Before we start, it is important to mention that the measurements were taken on a node that has 384 threads, however, only 200 threads where available. Another important detail concerns the problem size. In order to test the limits of the algorithm, the measures have been made on very big matrices (1024 $\times$ 1024 $\times$ 1024). This size is much larger compared to the size of the meshes that the algorithm will treat in reality. Making the measures on such a problem allows having more distinguishable differences between the programs, and helps to see if the program scales effectively.

Firstly, the plot 15 shows the computation of the two programs. As we can see, the Hedgehog program is faster than the FDS one. Furthermore, as shown by the standard deviation, it is also much more consistent. This demonstrates that using block decomposition improves the speed and the consistency of the program.

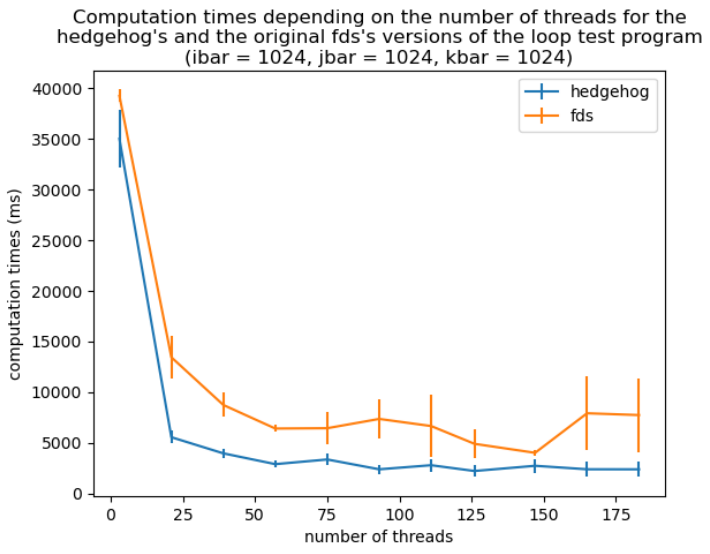

Figure 15: Computation times of the test program for Hedgehog and FDS (200 threads node)

Figure 16 shows the acceleration of the Hedgehog program compared to the FDS one. We can notice that on the point were FDS shows its best performance, Hedgehog is nearly two times faster. Interestingly, FDS starts to be very unstable with more than 150 threads, which results in a very high speedup of the Hedgehog program.

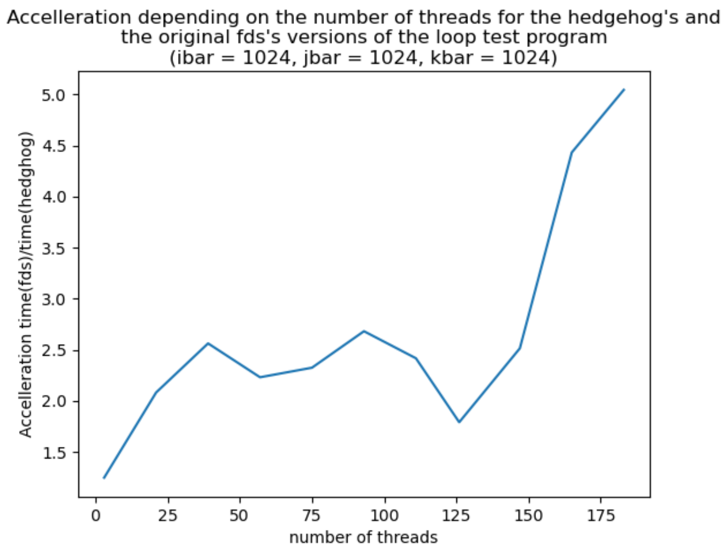

Figure 16: Acceleration of Hedgehog over FDS on the test program (200 threads node)

Finally, Figure 17 shows the relative speedup for each program. This relative speedup is not very high for both of the programs, indicating that we have not achieved sufficient parallelism [4] yet. For Hedgehog this can be explained by the fact that the chosen block size is not optimal. In these measurements, the block size was $(1024 \times 32 \times 1)$, which might be either too large or too small. Generally, finding the optimal block size is difficult as there are no definitive methods to determine it. The usual approach is to test the program with various block sizes. However, this was difficult to do on this program considering the fact that the initialization is very slow. It took more than twenty hours of computation to obtain the measurements presented in this section.

[4] In this context, achieving high degree of parallelism means utilizing all the resources of the processors.

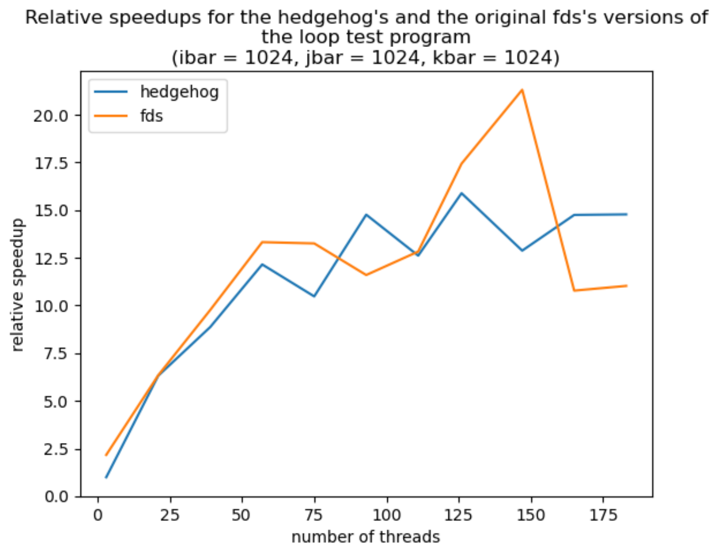

Figure 17: Relative speedup for Hedgehog and FDS on the test program (200 threads node)

Ultimately, these results demonstrate that the block decomposition can have a significant impact on the computation time. Furthermore, testing this program permitted to understand how to use Fortran and C++ together. In the next section, we will explain why this method cannot be applied to the global simulation.

3.2.4 Limitations

This method is effective and relatively easy to set up, however it still has some limitations.

Firstly, the fact that the data is entirely managed by Fortran poses a problem. Indeed, Hedgehog creates a data-flow graph, so it should be responsible for managing all the data. This would not have a significant impact on the performance, but it would improve the readability of the graph, as it would allow to clearly see which task access which data. Having a clear and well organized graph is very important to keep the application maintainable and expandable.

Secondly, with this method, it is not easy to have computation in C++. As discussed Section 2, some parts of the code were completely rewritten in C++. Keeping all the data as global variables in Fortran complicates the usage of pure C++ or external [5] sub-graphs. The data that flows into the global graph should not be defined in both languages, otherwise, this would require to have synchronisation tasks for copying the data between them. Such tasks would have a significant impact on the performance.

Finally, it may not always be possible to keep the types as they currently are in FDS. Indeed, as explained in Section 3.1, the simulator performs computation on meshes and this computation can be done on multiple nodes. Since these meshes belong to the same geometrical area, after each computation, we need to synchronise the modifications made to the stencils [6] of the meshes. This problem will also be present with the blocks on the Hedgehog version. To perform the update efficiently, we will need to have a data structure that will represent the stencils, which does not exist in the FDS code yet. Moreover, it is preferable not to implement this data structure in Fortran. Firstly because it might be more difficult to implement (Fortran does not have all the tools that C++ has), and secondly because the code that will perform the update will be easier to write in C++.

[5]Here, external means that the sub-graphs uses other languages.

[6]Area around a mesh that is shared with other meshes

Working on this example program was very useful, primarily as it helped to understand some parts of the code. Furthermore, this was a good demonstration of what could be done with Hedgehog on the FDS code, and it allowed us to experiment with a part of the code extracted from the whole simulation. Unfortunately, the method that has been used here cannot be utilized for parallelizing the rest of the program, but we learned a lot from this exercise.

3.3 Conclusion

So far, we have seen the work that has been done on different parts of the simulation. These parts have been treated separately to try different methods for optimizing the entire code. We have demonstrated that it is possible to rewrite some sections entirely in C++, such as the Cholesky decomposition. We also experimented using Hedgehog as an orchestrator while keeping all the computation and the memory in Fortran, and we have seen why this is eventually not the ideal approach.

To parallelize the entire simulation with Hedgehog we have tried to tackle the problem from two sides. First, we aimed to parallelize specific pieces of the application, as we did for the velocity. Second, we focused on creating a global graph and try to set up a pipeline threw the entire simulation. The role of the global graph is to call Fortran subroutines. The gaol is to refactor the graph in order to have more specific tasks after each iteration. For instance, we begin with a graph that calls a main function in Fortran. Then we refactor this graph into a new one that has three tasks: the initialization, the main loop and the end of the program. Then we try to split each of these three tasks into sub-tasks to create a new graph, and we keep iterating till our tasks are precise enough. Eventually, we should end up having atomic tasks, and some of them can be run in parallel.

This approach was logical, however, implementing everything manually is difficult due to the complexity and the size of the code base (more than 160K lines of code). Furthermore, doing this manually is not a method that is easily reproducible. For this reason, we have decided to try building a tool able to perform some automatic transformations on the code. The specifications of the tool will be described in the next section.

4 Automatic graph generation

In Section 3.3, we discussed that porting FDS to Hedgehog was a very difficult task that could not be done manually. Furthermore, this was not a method that could be easily used on even bigger project. Fortunately, while parallelizing different algorithms with Hedgehog, we observed some common patterns. For instance, the operation of splitting the code into simple functions, that can be bound to tasks in the graph, is done multiple times. Another common pattern is analysing the data used by a function or an algorithm, to determine the form of the data that will flow into the graph. Based on these observations we wondered if it would be possible to create a tool that would be able to perform these operations automatically. To answer this question, we have initiated a project in collaboration with *Semantic Designs*, a company that has a lot of experience with the technologies we plan to use for our tool. In this section, we will briefly describe the specifications of the tools as well as some ideas that we currently have for its implementation. We will not go further into the details as we are still in the early stages of the project.

4.1 The specifications

The purpose of the tool is to translate the code into C++ and then refactor it and build a graph. There are two reasons for translating the code into C++ before modifying it. Firstly, this approach will enable us to work with any language. Indeed, transformations on the code can vary depending on the language, so it will be easier to work only with C++. Secondly, there is a technical limitation of the compilers imposed by the C standard. In fact, FDS manipulates many data, and some of the subroutines use a lot of variables. Currently, the variables are global, however, managing the memory in C++ will require to pass these variables as parameters. According to the C standard [9], there can only be a maximum of 127 parameters in a function, which could be a limitation in our case[7]. The only way to pass variables to Fortran subroutines is by using pointers on fundamental types. It is not possible to bind a Fortran type to a C structure that contains pointers [20]. Because of this limitation, if a subroutine needs to access more than 127 elements that are not fundamental (arrays for instance), it will not be possible to call this subroutine from C++.

After translating the source code into C++, the tool must be capable of making some changes to the code. For instance, it should remove all global variables automatically. This task is complex because there are a lot of things consider (is a variable used elsewhere, should it be local anyway[8], ...). Another operation is reorganizing the code into functions. Most importantly, this reorganization must be done in a way that allows parallelism. Additionally, after the modifications, the tool should generate the Hedgehog graph automatically. This involves even more complexity because besides of generating the tasks, we also have to generate the types of the data that will flow into the graph.

The tool should also have some additional secondary features. We plan to introduce special comments to the code, using a syntax similar to doxygen, to guide the tool on how to reorganize the code. For instance, we might use a *task* or a *graph* comment, to indicate that a certain section of the code should belong to a task or a sub-graph. Finally, we intend to create a custom language for automating some operations on the code (extract a function for example). The objective is to train a LLM (Large Language Model) to generate these operations.

Now that we have defined the specifications of the tool, shall we talk about the implementation.

[7]This is the C standard and not the C++ one since the functions that are bind to the Fortran subroutines are `extern "C"`.

[8]In FDS, loops variables can be global.

4.2 Implementation

Implementing this tool will require a transpiler to parse the Fortran code and translate it in C++. We will then modify the C++ AST of the program in order to perform the operations that we described earlier. The code will be transformed in a manner similar to that used by compilers, however, the goal is not to optimize the code but to generate a Hedgehog graph. Fortunately, *Semantic Designs* already have some libraries and tools that can be used for transpiling the code and change the AST. However, their parser will be adapted in order to support the new comment syntax.

This tool will be a very interesting and useful project that will utilize the compilers technologies in an innovative way. Initially, we will focus on generating code for Hedgehog, but other directions may be explored in the future.

5 Serialization library

In this section, we will present the optimizations added to a serialization library originally developed as a school project at ISIMA [10]. The purpose of these optimizations is to make this library usable in HPC applications. In our context, the objective was to use this library to transfer data between nodes using MPI in the cluster version of Hedgehog. We will also describe some of the new features of the library.

5.1 The principle

First, we will explain the principles behind the first version of the library. We will not go too deeply into details, as these have already been described in [10].

The basic purpose of the library is to enable serialization of C++ classes by writing as little code as possible, similar to the use of decorators in languages such as Java. The library provides macros that are used in the serialized classes. These macros generate the `serialize` and `deserialize` functions in the class. Listing 8 illustrate how to make a class serializable, using the interface from the first version of the library. The `SERIALIZABLE` macro generates the serialization functions, and the `SERIALIZER` macro initializes the *serializer*.

```cpp
#include <serializer/serializable.hpp>

class MyClass {
    SERIALIZABLE(int, double);
  public:
    MyClass(int x, double y): SERIALIZER(x_, y_), x_(x), y_(y) {}

  private:
    int x_ = 0;
    double y_ = 0;
};
```

Listing 8: Example of a class serialization

The *serializer* is an attribute added to the class by the `SERIALIZABLE` macro. It contains a data structure, the `AttrContainer`, which stores references to the attributes of the serialized class, in order to be able to access their values or modify them. This data structure functions similarly to a list, in which each node can store a different data type. It can also be compared to a `tuple`. In the generated serialization function, we call the methods of the *serializer* to initiate the serialization of the attributes in the container. This mechanism has changed in the latest version of the library, which will be explained in Section 5.2.

To serialize the attributes, we use a specialized class called `Convertor`. This class contains very generic template functions capable of converting most of the types of the standard library, as well as some external types if they meet certain conditions. This functionality means that the library users do not need to translate the serialized members by hand, everything is done automatically in the *convertor*. Additionally, it is possible to create a custom *convertor* class, either to add support for external types or to serialize polymorphic objects. The class has been changed since the first version of the library, and these modifications will be explained in

Section 5.4.

Now that we have explained the fundamental principle of the first version of the library, we will now detail the optimizations and the features that have been added to it, as well as the new interface.

5.2 The new interface

The interface of the first version of the library had several significant limitations, and was not as simple as it should have been. The new interface is even simpler and solves the issues present in the first one.

The previous interface required the use of two macros that were generating a significant amount of code. The macro `SERIALIZABLE` added a *serializer* attribute to the class. As a result, the users had to implement their own constructors rather that using those automatically generated by the compiler. Consequently, a lot of unnecessary additional code had to be written to manage the *serializer* attribute.

Another issue of the old interface was the extensive use of intermediate layers. To serialize a class, we had to call the functions generated by the macros within the class. These functions where using methods from the *serializer* that were using the attribute container that was calling the *convertor* methods. The usage of all these functions had a big impact on the performance. Furthermore, it appeared that the `AttrContainer` was inefficient because of the recursive calls.

The new interface does not use the `AttrContainer` and the *serializer*. Furthermore, only one macro, `SERIALIZE`, is used to generate the serialization functions. Listing 9 illustrates the usage of the new macro.

```cpp
class MyClass {
  public:
    MyClass(int x, double y) x_(x), y_(y) {}

    // the class is now serializable
    SERIALIZE(x_, y_);

  private:
    int x_ = 0;
    double y_ = 0;
};
```

Listing 9: Serializing a class with the new interface

The generated serialization functions now use external functions that directly invoke the *convertor*. Listing 10 shows the signature of the `serialize` function. The external function is used rather that generating all the code inline as it simplifies maintenance.

```cpp
template <typename Conv, typename... Args>
constexpr void serialize(std::string &str, type_list<Args...>,
                         tools::mtf::serialize_arg_type_t<Args>... args);
```

Listing 10: External `serialize` function

5.3 Binary serialization

The first version of the library was generating a human-readable string with a format similar to JSON. The new version uses the binary format instead. The binary format is more efficient since it takes less space and is very easy to parse. Moreover, it allows the removal of a lot of unnecessary information. For instance, the previous format required to know the identifiers of the member variables of the class. The new format assumes that the data will always be serialized and deserialized in the same order, eliminating the need for the identifiers. Furthermore, the previous format required the use of separators to identify the beginning and the end of objects, containers and strings (that had to be escaped manually[9]).

To serialize fundamental types in binary, we use `std::bit_cast` to convert the address of the serialized variable to `char*` and vice-versa. For the containers and strings, the process begins by writing the number of elements (on a `size_t`), followed the elements themselves. When serializing a pointer, we first add a character (either `v` or `n`) to specify if the pointer is valid or null. If the pointer is null, there is nothing else to add. Otherwise, we deserialize the element on which the pointer points to. During the deserialization, the pointers are dynamically allocated if the variable is null, otherwise, the pointed value is modified directly. This is important, especially for dynamic arrays since the dynamic allocation is a slow process. Pointers can lead to segmentation error or memory leaks if they are not handled properly. In our situation, there are no solutions to prevent such behaviors, so we assume that the user of the library do things well or do not use pointers.

5.3.1 Example

To visually explain how the data are organized in the serialized string, let us consider an example. We will use 2 classes, `MyClass` shown in listing 11 and `Class2` defined in listing 12.

The class `MyClass` has five attributes. The first two attributes have fundamental types `int` and `double`. The third attribute is a vector of integers, followed by a pointer to a double. The last attribute is a shared pointer on a `Class2` object.

[9]The string were represented between `'"'`, so we had to escape potential the double quote character inside the string in order for them not to be considered as a separator during the deserialization.

```cpp
struct MyClass {
  int i;
  double d;
  std::vector<int> v = {};
  double *ptr = nullptr;
  std::shared_ptr<Class2> s_ptr = nullptr;
};
```

Listing 11: Definition of MyClass

The class `Class2` has only one attribute which is a string.

```cpp
struct Class2 {
  std::string id;
};
```

Listing 12: Definition of Class2

An object `MyClass` is instantiated with the values that are described in the table 1.

Attributes	Values
i	14
d	1.3
v	{ 1, 2, 3 }
ptr	nullptr
s_ptr	@address
s_ptr.id	"name"

Table 1: Values of the attributes in the object `myClass`

If we try to serialize this object, we will obtain the following output:

```
7 MyClass 14 1.3 3 1 2 3 n v 13 Class2 4 name
[7: MyClass] [14] [1.3] [3: 1, 2, 3] [n] [v: [13: Class2] [4: name]]
```

The first line is the content of the final string (the values in binary have been translated). On the second line the different parts of the string have been regrouped between square brackets. The string always begins with the identifier of the type of the serialized class. For instance, an object of type `MyClass` begins with *MyClass*. The name of the type is the only meta-data stored in the serialized string. It should be used to construct the right object when we only have the serialized string (when we deserialize a polymorphic type for instance). Additionally, the content of the containers is always preceded by the number of elements. For instance, `v` contains three elements, so its content is preceded by `3`. Finally, the null pointers are identified with the letter `n` and the valid ones with the letter `v` followed by the content as shown for the attribute `s_ptr`.

5.3.2 C-Struct optimization

It is possible, to use `std::bit_cast` on structs directly, which can be very fast. However, determining when this can be done is difficult. Indeed, we cannot directly cast a structure that contains pointers or complex data structures, such as vectors, this only works with C-Structs (structures with only basic types). With the new interface, we could make the test by iterating on the list of the types of the serialized members. However, in this case, there is no guaranty that the user wants to serialize all the data in the class. There may be members with complex types that are not serialized, and attempting to cast the structure would not be optimized. The easiest way to know if it is possible to use the technique is by letting the user decide on its own. A dedicated macro has been added in the new interface.

5.4 Redesigning the convertor

Since most types are serialized automatically, it was required for the user to have a way to manually add conversion functions for external types. Indeed, the users do not always have the possibility to make classes serializable. For instance, with the Qt framework, we do not have access to the class `QString` and this class cannot be serialized by the functions of the default convertor.

In the previous version of the library, all the convertor functions were defined in the macro `CONVERTOR`. This macro had to be used at the end of each convertor class. This was done because the custom conversion functions needed to be defined before the default ones in order for the code to compile. To understand why basic inheritance could not be used directly, we will consider the following example: the user wants to serialize a class that holds a vector of an `Unknown` type. This type is an external one, and it is not supported by default, so it requires implementing a custom convertor using inheritance. Now, we are going to try to compile the code. During compilation, when serializing a vector, the dedicated conversion function is in the base class of the custom convertor. However, when converting the vector, we need to invoke a function that handles the elements within the vector. This function defined in the subclass is inaccessible from the base class. One solution would be to use virtual functions. However, it was not possible by that time because all the conversion function were templates as SFINAE (Substitution Failure Is Not An Error) was used to differentiate one function from another.

Using such a macro in the library was a significant problem. Indeed, the first issue is the difficulty of maintaining the code. Because all the functions were defined in the macro, the compiler errors were difficult to read, and it was not possible to use the full power of an IDE. The second issue is the fact that a lot of code was generated. In fact, if the user wanted to create multiple convertor classes, there was a lot of code duplication. Finally, the usage of such a macro from the user's point of view was very unintuitive.

To remove the `CONVERTOR` macro, a method similar to the behaviors of Hedgehog was utilized [2]. The `Convert` behavior is a template class with two virtual pure functions, `serialize` and `deserialize`. These functions should be overridden in the user's convertor class. The fact that the class is template but not the methods allows them to be virtual. The definition of this class is shown in Listing 13.

```cpp
template <typename T> struct Convert {
    virtual void serialize(T const &, std::string &) const = 0;
    virtual void deserialize(std::string_view &, T &) = 0;
};
```

Listing 13: Convert class

The default convertor class has non-static member functions `serialize_` and `deserialize_` (an `_` is used to avoid conflicts with the functions from `Convert`). These methods are templates, and they use concepts from C++20. The usage of concepts is very important in this case because they are more flexible than SFINAE. In fact, they easily allow the definition of fallback functions that are called when a non-supported type is serialized. These fallback methods use the functions from `Convert` as we can see in Listing 14. In these functions, exceptions are used to create useful error message at runtime. The choice of generating errors at runtime and not at compile time (using `static_assert`) was made because the error messages are clearer, and more readable as they are not lost in the compiler output.

```cpp
  template <serializer::tools::concepts::NonSerializable T>
  void serialize_(T const &elt, std::string str) const {
      if constexpr (tools::mtf::contains_v<T, AdditionalTypes...>) {
          // we need a static cast because of implicit constructors (ex:
          // pointer to shared_ptr)
          static_cast<const Convert<T> *>(this)->serialize(elt, str);
      } else {
          throw serializer::exceptions::UnsupportedTypeError<T>();
      }
  }
```

Listing 14: Fallback serialization method

The last important thing is the fact that the default convertor inherits from multiple `Convert` classes (one per new type added). To achieve that, we can see in Listing 15 that we use the fold expressions with a variadic template.

```cpp
template <typename... AdditionalTypes>
struct Convertor : Convert<AdditionalTypes>... { /* ... */ };
```

Listing 15: Convertor class

The new `Convertor` class is then easier to use than the old macro. It is also easier to maintain since macros in C and C++ are not made to contain such an amount of code. Lastly, with this version, there is less code duplication which makes this library usable in a case where memory is limited (embedded C++ applications for instance).

5.5 New features

In addition to the optimization, some new features have been implemented. They will be presented here.

5.5.1 Static arrays

A conversion function has been implemented for static arrays. In order to serialize arrays, we need to know their sizes. For this, we use `std::extent_v`. For example, `std::extent_v<T[N]>` returns `N`. The conversion function is recursive and uses `std::remove_extent_t` to convert multidimensional arrays. Listing 16, illustrate static arrays serialization.

```cpp
class MyClass {
  public:
    /* ... */
    SERIALIZE(serializer::type_list<int[10], int[10][10]>(), arr_, grid_);

  private:
    int arr_[10];
    int grid_[10][10];
};
```

Listing 16: Example: serializing static arrays

It is possible to use the `type_list` to specify the template parameters in the macro. This has to be done when the compiler cannot deduce the types automatically or when the functions and dynamic arrays are used as we will see in Sections 5.5.2 and 5.5.3.

5.5.2 Dynamic arrays

To be serialized, dynamic arrays need to be differentiated from the standard pointers. Furthermore, the serialization also requires their size, which might not be known at compile time. To solve these problems, a wrapper type `DynamicArray` was implemented. The definition of this class is shown on Listing 17.

```cpp
template <concepts::Pointer T, typename DT, typename... DTs>
struct DynamicArray {
    explicit DynamicArray(T &mem, DT dim, DTs... dims)
        : mem(mem), dimensions(dim, dims...) {}
    explicit DynamicArray(T &mem, std::tuple<DT, DTs...> dimensions)
        : mem(mem), dimensions(dimensions) {}
    T &mem;
    std::tuple<DT, DTs...> dimensions;
};
```

Listing 17: `DynamicArray` class

The DynamicArray class stores a reference to the member variable holding the array's pointer. This reference is used to facilitate the change of the pointer (potential allocation or free on deserialization). The dimensions of the array are stored in a tuple. The types of the dimensions are configurable, allowing users to pass references to member variables holding the dimensions, as well as values if the size is known at compile time. The number of dimensions is not related to the dimensions of the pointer. Users can have multidimensional arrays stored in a 1D pointers, and specify all the dimensions to the DynamicArray. For instance, we can have the following type: `DynamicArray<double*, size_t&, size_t>`. In this example, a 2D array (matrix) stored in a 1D pointer. The first dimension of this matrix might be variable, so it is stored in a member variable given by reference to the DynamicArray. The second dimension is constant, so it is passed by value.

Listing 18 shows an example of serializing a class with dynamic array attributes. In this example, the macro SER_DARR_T is used to generate the type of the dynamic array. For instance, `SER_DARR_T(types...)` is expanded to `serializer::tools::DynamicArray<types...>` (the macro avoids writing the namespaces). The class MyClass holds two dynamic arrays: arr1D_ and arr2D_. The size of arr1D_ and *arr2D_ is known at compile time (5 and 2), whereas the size of arr2D_ is known at runtime (only when allocArr2D is called). The size of arr2D_ is then stored in a variable that is serialized and that is taken by reference by the DynamicArray constructor as explained earlier.

```
class MyClass {
  public:
    explicit MyClass() : arr1D_(new int[5]) { }
    ~MyClass() { /* free memory */ }
    void allocArr2D(size_t arr2DHeight) { /* ... */ }
    /* ... */
    SERIALIZE(serializer::type_list<size_t, SER_DARR_T(int *, size_t),
                  SER_DARR_T(int **, size_t &, size_t)>(),
              arr2DHeight_, SER_DARR_T(int *, size_t)(arr1D_, 5),
              SER_DARR_T(int **, size_t &, size_t)(arr2D_, arr2DHeight_, 2));

  private:
    int *arr1D_ = nullptr;
    int **arr2D_ = nullptr;
    size_t arr2DHeight_ = 0;
};
```

Listing 18: Example: serializing dynamic arrays

5.5.3 Functions

Finally, it is now possible to execute functions at any stage of the serialization or deserialization. Functions can be used to handle specific cases. For example, they can be used to serialize complex objects with cyclic references. Indeed, the library is not able to detect if a pointer or a reference has already been serialized. It would be possible to perform this verification, however, this would impact the performance. Now, to do that, it is possible to use a function that will either create a link during the deserialization (link the child to it's father in a tree) or perform more complex operations (coloring nodes when serializing a cyclic graph). The functions can be defined globally or inside the class using a lambda expression or a static function.

From the library's developer point of view, these functions can be used for debugging. In fact, as we have already explained, the attributes are serialized in the order they appear in the macro parameters, and this is also the case for functions. For instance, functions can be used to print the values of each member, as well as the value of the serialized string, before and after the members are serialized.

It is not possible to use any function, for now, only the type `function_t` is accepted. Its definition is shown in Listing 19.

```
using function_t = std::function<void(Phases, std::string_view const &)>;
```

Listing 19: Serializer function type

The functions take two parameters:

- The phase that is either `Phases::Serialization` or `Phases::deserialization`. It is used to know if we are serializing or deserializing.
- A constant string view that is either the result of the serialization or the deserialized string depending on which phase is executed. For now, we consider that the user should not be able to modify this parameter, but this could change in future version (we will study more use cases of the functions to make this choice).

It is important to mention that the type `function_t`, as well as the type `DynamicArray`, are not taken by reference in the `serialize` function. This is due to the fact that we want to be able to create the objects directly in the `SERIALIZE` macro. To achieve this, a dedicated meta-function as been written, shown in Listing 20.

```
template <typename T> struct arg_type { using type = T &; };

template <> struct arg_type<function_t> { using type = function_t&&; };

template <typename T, typename DT, typename... DTs>
struct arg_type<serializer::tools::DynamicArray<T, DT, DTs...>> {
    using type = serializer::tools::DynamicArray<T, DT, DTs...>;
};
```

Listing 20: Usage of `arg_type_t`

Listing 21 illustrate how to serialize a node inside a binary research tree. In this example, a node has four attributes: the value stored in the node, the left and the right children, and the father node. In this case, we want to save the link between the child node and the father node, however, it is not possible to serialize the father node directly since it is a cyclic reference. Instead, we use a function to recreate the pointer link at the end of the deserialization. Note that once more, a macro is used to simplify the syntax. There are three helper macros for functions:

- `SER_SFUN`: generates a lambda that is executed during the serialization.
- `SER_DFUN`: generates a lambda that is executed during the deserialization.
- `SER_FUN`: generates a lambda that is executed during both phases.

Since macros substitute text, we still have access to the parameters of the functions, even if they are not visible. The phase parameter is named `phase` and the string view parameter is named `str`.

```cpp
template <typename T>
struct Node {
  public:
    explicit Node(T value = 0): value(value) {}
    ~Node() { /* ... */ }

    /* ... */

    SERIALIZE(
        serializer::tools::mtf::type_list<T, Node<T> *, Node<T> *,
                                          serializer::function_t>(),
        value, left, right, SER_DFUN({
            if constexpr (!std::is_const_v<
                              std::remove_pointer_t<decltype(this)>>) {
                static_assert(
                    !std::is_const_v<std::remove_pointer_t<decltype(this)>>);
                if (this->left)
                    this->left->father = this;
                if (this->right)
                    this->right->father = this;
            }
        }));

    T value;
    Node<T> *father = nullptr;
    Node<T> *left = nullptr;
    Node<T> *right = nullptr;
};
```

Listing 21: Example: using a function for serializing a tree node

5.6 The performance

The performance of the library have been drastically improved compared to the first version. However, the serialization remains slightly slower compared to other libraries such as *cereal* or *cista*. Comparisons between the libraries have been made using a benchmark repository available on GitHub: `felixguendling/cpp-serialization-benchmark`. The benchmark creates and serialize a graph with more that 18k nodes. Listing 22 shows the structure that was used in the benchmark.

```cpp
struct serializer_graph {
  SERIALIZE(serializer::type_list<std::vector<node>>(), nodes_);

  struct edge {
    SERIALIZE(serializer::type_list<uint16_t, uint16_t, uint16_t>(),
              from_, to_, weight_);
    uint16_t from_, to_;
    uint16_t weight_;
  };

  struct node {
    SERIALIZE(serializer::type_list<uint16_t, std::string,std::vector<edge>,
                std::vector<edge>>(), id_, name_, out_, in_);
    uint16_t id_;
    std::string name_;
    std::vector<edge> out_ = {};
    std::vector<edge> in_ = {};
  };
  /* ... */
  std::vector<node> nodes_ = {};
};
```

Listing 22: Graph structure of the benchmark

As shown by the results, the serialization is slower that most of the other libraries. However, in terms of memory usage, the library performs well.

```
--------------------------------------------------------------------------
Benchmark                  Time              CPU   Iterations UserCounters...
--------------------------------------------------------------------------
capnp                    997 ms           997 ms            1 size=50.5093M
cista_offset_slim       83.1 ms          83.1 ms            7 size=25.317M
cereal                  1109 ms          1108 ms            1 size=37.829M
serializer              1728 ms          1728 ms            1 size=37.829M
fbs                    11369 ms         11367 ms            1 size=62.998M
cista_raw               7789 ms          7788 ms            1 size=176.378M
cista_offset            8278 ms          8277 ms            1 size=176.378M
zpp_bits                16.3 ms          16.3 ms           39 size=37.8066M
```

These are the results for the deserialization. We can see that the deserialization is relatively efficient compared to the serialization.

```
----------------------------------------------------------
Benchmark                 Time             CPU   Iterations
----------------------------------------------------------
capnp                   0.003 ms        0.003 ms       227360
cista_offset_slim        37.8 ms         37.8 ms           19
fbs                      1386 ms         1385 ms            1
cista_offset             2512 ms         2512 ms            1
cista_raw                2346 ms         2346 ms            1
cereal                   1007 ms         1007 ms            1
serializer               1004 ms         1004 ms            1
zpp_bits                 11.3 ms         11.3 ms           56
cista_raw                1196 ms         1196 ms            1
```

It is possible to drastically improve the performance by using the macro `SERIALIZE_STRUCT` in the `edge` type. This macro performs a `bit_cast` directly on the object, without using the convertor. This optimization reduces the serialization and deserialization times to around 600 milliseconds, making the library faster than *cereal*. It demonstrates that the convertor remains relatively slow, and more optimizations should be done to it.

Some more optimizations will be done in the future to improve the serialization. Moreover, binary serialization itself might not be enough to obtain good performance, and should be used alongside other techniques. For instance, some libraries rely on alignment to optimize the size of the result binary stream.

5.7 Conclusion

In this section, we have described the concept behind the serialization library, and we have explained how it has been optimized. The usage of binary serialization and simpler data structures that do not involve many pointers such as the strings allowed us to improve the performance compared to the previous version of the library. Furthermore, the code has been improved by removing the *CONVERTOR* macro. Some new features have also been added to enhance the capabilities of this tool. Static and dynamic arrays are now serializable, and we can also execute code during the serialization or the deserialization. Finally, even more optimizations should be added in the future to compete with the other libraries available on the market.

Part III
Results and discussion

1 State of the project

In this section, we will resume the work that was done on FDS and the serialization library, then we will explain what remains to be done.

1.1 FDS

The project of porting FDS to Hedgehog is not done yet. We began by implementing critical sections of the code such as the Cholesky decomposition or velocity computation. Then we tried to analyse and reorganize the simulation code. However, we eventually concluded that, due to the amount of work, this process should not be done manually. The FDS project was then paused, and we began the design of a tool that we could be used to automate the generation of a Hedgehog graph. The work completed on the simulation is important because it will help us to make choices in our future work.

The tool that will automate the generation of the graphs has not been implemented yet. We currently are in the early stages of the design. However, the initial research and ideas we have, are promising. The problem is definitely not trivial, and it may take several months before we can perform even basic transformations on the code.

1.2 The serialization library

The serialization library has been optimized compared to the previous version and now includes some additional features. There will be more optimization that will be added in the future since the serialization is still slower compared to the other libraries available on the market. Additionally, the new interface is even simpler and more flexible, allowing the user to serialize classe by using only one macro.

2 Planning

In this section, we will compare the expected planning with the real one by commenting two Gantt charts.

Figure 18 shows the expected Gantt chart for this project. This chart indicates that the first month was dedicated to the learning of Hedgehog. The second task was the implementation of the Cholesky decomposition (the chart has been made when this task began; it corresponds to our first meeting at the NIST). Then, the other tasks correspond to the port of FDS to Hedgehog. Originally it was planned to work exclusively on FDS and to complete the project before the end of the internship. Unfortunately, as we saw earlier, we eventually decided to adopt another method in the middle of the project.

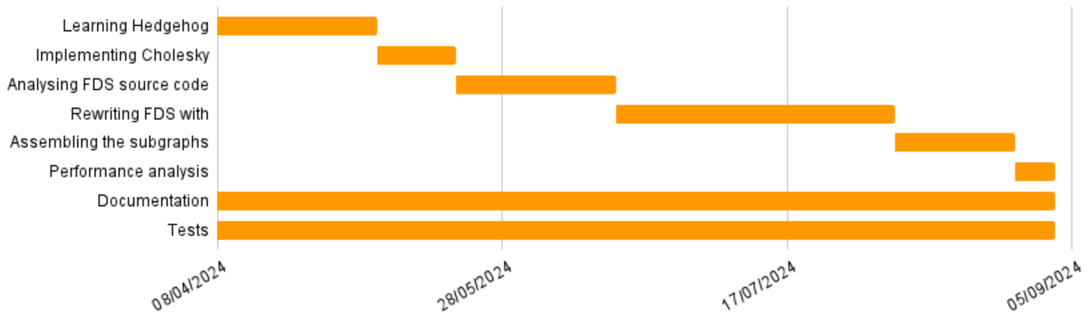

Figure 18: Expected Gantt chart

On the real Gantt chart, shown in Figure 19, the tasks of learning Hedgehog and implementing the Cholesky decomposition are still there. However, there are new tasks on this second chart. Firstly, there is the work on the 3D loops program. When we started the project, we did not have the knowledge of this test program. We decided to work on it after the FDS's developers presented it to us. Then, as we have seen in the report, optimizing FDS by hand was too complicated, so we decided to implement a tool that could automate a part of the task. Finally, since it has been proposed to use the serialization library during the internship, this task only appears in the real Gantt chart.

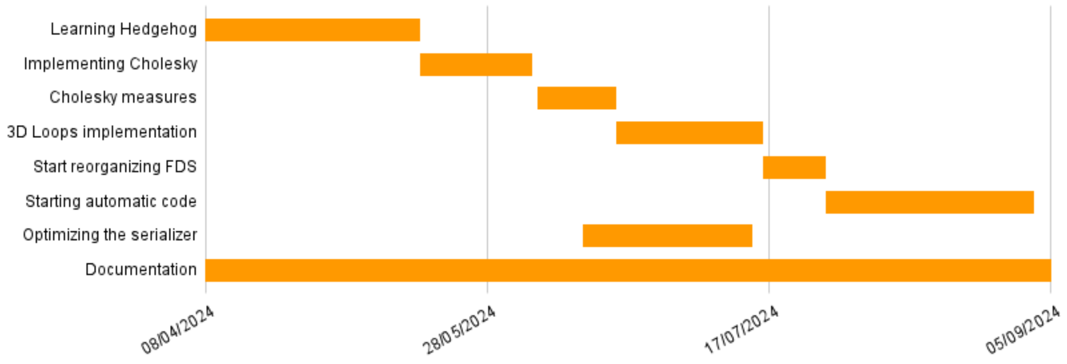

Figure 19: Real Gantt chart

Those diagrams are very different for two reasons. Firstly, it is difficult to evaluate the work on an unknown code base such as the FDS's one. Secondly, the usage of the serialization library and the implementation of the refactoring tool were propositions that were made during the project, with some knowledge and observations that we did not have originally.

3 Discussion and prospect

In ths serction we will discuss the results that we have obtained and the way we have obtained them. We will also talk about the future of the project.

3.1 FDS

When we started working on FDS, we thought it was possible to port the code to Hedgehog manually. Some of the critical algorithms of the simulation have been rewritten with our library, and we managed to have good results. However, we eventually came to the conclusion that doing everything by hand was not the best method. Firstly, this was a very difficult task considering the size of the FDS's code base and the large number of data that is used in the program. Secondly, this method is not reproducible and cannot be applied on other projects. Even if some tasks are technically feasible, it is always important to think about the scalability and the reusability of the methods we use. One of the most important roles of a software engineer is to formulate the method he used to solve a problem, in order to automate it and reuse it later on.

Currently, the simulation has not been optimized with Hedgehog yet. However, it may be in the future, when the refactoring tool will be operational.

3.2 The serialization library

The serialization library is a very interesting project and the library as a unique design. The library has currently a lot of features. However, despite the optimizations implemented, it is still slower that the other libraries on the market. It is definitely usable with Hedgehog, but it would be interesting to improve the performance even more.

Another potential issue for some developers is the heavy use of macros. Many users dislike macros in C++ because they hide a lot of generated code. Additionally, the preprocessor does not perform any verification when it generates the code, which increases the probabilities of errors. Providing an even simpler interface for using the library without the macros could be beneficial to such users.

4 Environmental analysis

In this section, we will try to analyze the environmental impact of the project.

The context of the project was the research on High Performance Computing. The HPC concept is not ecologic by itself. The purpose of this domain of research is to ensure that applications utilise a maximum of the resources of computers. These applications are usually executed on very energy-intensive clusters. However, in some cases, HPC can be used ecologically.

Firstly, when a lot of computing is required, HPC research can be important, not to reduce the energy consumption of clusters, but use the energy efficiently. In fact, if no specific optimizations have been done on a computer, as soon as the processor is powered on, it consumes energy, even if only half of its cores are effectively used. Thanks to HPC techniques, there are ways to utilise this energy more efficiently, by using all the available resources to accelerate the computation when it is possible.

Secondly, if HPC itself is not directly linked to ecology, but the techniques developed in the domain can be used to improve environmental research. For instance, simulations such as FDS are used a lot to predict environmental events and help to find solutions.

Finally, a lot of researches are conducted to optimize the energy consumption of the supercomputers. Indeed, researchers have found out that it was possible to control the power usage within processors. Interestingly, the fact of reducing the power in a processor when it is not fully used can also have a positive effect on the performance. [11] presents some very interesting work on power measurement and optimization. These techniques are used by the super-computers member of the *green500* list [21].

Conclusion

The objective of the project was to optimize the fire simulation FDS using Hedgehog. Another goal was to contribute to the development of Hedgehog by optimizing a serialization library originally created for a project at ISIMA.

The port of FDS to Hedgehog is not yet complete, as we concluded that this task was too complicated to be achieved manually. Therefore, we eventually decided to develop a tool that could automate a part of this process. However, we are still in the early stages of this project and no implementation has been done yet. Even if we stopped working directly on FDS, we have optimized some parts of the simulation. This preliminary work was useful to understand how the simulator works. Moreover, it had a significant impact on the specifications of the tool that we want to make.

During the internship, it has been proposed to use the serialization library in the cluster version of Hedgehog. New features have been implemented, and optimizations have been made to accelerate the serialization. The library offers a unique concept and provides a lot of tools. Despite these optimizations, it remains slightly slower than the other libraries on the market. However, it is more flexible because it is not restricted to the types of the standard library, and it is easily expandable. More optimization will be implemented in the future to compete with the other libraries.

This internship was a good introduction to the world of HPC. A lot of knowledge has been acquired, and an innovative project have been initiated. There are a lot of differences between the initial expectations and what was done eventually. However, the work in a laboratory often differs from the work in a company, as it is not always possible to predict the results when we start working on a research project.

Bibliography

[1] T. J. Blattner, W. Keyrouz, J. Wu, and S. S. Bhattacharyya, “Model-based dynamic scheduling for multicore implementation of image processing systems,” 2017.

[2] A. Bardakoff, “Analysis and execution of a data-flow graph explicit model using static metaprogramming,” Ph.D. dissertation, Université Clermont Auvergne, 2021.

[3] Kavi, Buckles, and Bhat, “A formal definition of data flow graph models,” *IEEE Transactions on computers*, vol. 100, no. 11, pp. 940–948, 1986.

[4] K. McGrattan, S. Hostikka, R. McDermott, J. Floyd, C. Weinschenk, and K. Overholt, “Fire dynamics simulator user’s guide,” *NIST special publication*, vol. 1019, no. 6, pp. 1–339, 2013.

[5] T. Korhonen and S. Hostikka, “Fire dynamics simulator with evacuation: Fds+ evac,” *Technical Reference and User’s Guide. VTT Technical Research Centre of Finland*, 2009.

[6] G. P. Forney and K. B. McGrattan, *User’s Guide for Smokeview Version 4: A Tool for Visualizing Fire Dynamics Simulation Data.* US Department of Commerce, National Institute of Standards and Technology, 2004.

[7] K. B. McGrattan, H. R. Baum, R. G. Rehm, *et al.*, *Fire dynamics simulator–Technical reference guide.* National Institute of Standards, Technology, Building, and Fire Research Laboratory, 2000.

[8] J. H. Ferziger, “Large eddy simulation,” *Simulation and modeling of turbulent flows*, vol. 109, p. 154, 1996.

[9] I. ORGANISATION, *Iso/iec 9899: 1999 programming languages-c*, 1999.

[10] R. C. Théo CLIQUOT. “Creation d’une application de jdr et conception d’une bibliothèque de sérialisation automatique en c++,” Institut Supérieur d’Informatique, de Modélisation et de leurs Applications (ISIMA). (2024).

[11] S. Song, R. Ge, X. Feng, and K. W. Cameron, “Energy profiling and analysis of the hpc challenge benchmarks,” *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 265–276, 2009.

Webography

[12] Wikipedia contributors, *National institute of standards and technology — Wikipedia, the free encyclopedia*, [Online; accessed 5-August-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=National_Institute_of_Standards_and_Technology&oldid=1234312240.

[13] Graphviz community, *Graphviz*, [Online; accessed 14-August-2024], 2021. [Online]. Available: https://graphviz.org/.

[14] Alexandre Bardakoff, Timothy Blattner, *Hedgehog tutorials: Tutorial 1*, [Online; accessed 5-August-2024]. [Online]. Available: https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial1.html.

[15] Wikipedia contributors, *Cholesky decomposition — Wikipedia, the free encyclopedia*, [Online; accessed 30-June-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cholesky_decomposition&oldid=1219502782.

[16] S. Ostrouchov, *Cholesky Factorization.* 1995. [Online]. Available: https://www.netlib.org/utk/papers/factor/node9.html.

[17] Kevin B. McGrattan, Randall J McDermott, *Fds and smokeview*, [Online; accessed 6-August-2024], 2021. [Online]. Available: https://www.nist.gov/services-resources/software/fds-and-smokeview.

[18] R. J. M. Kevin B. McGrattan, *What is fds?* [Online; accessed 6-August-2024], 2024. [Online]. Available: https://fdstutorial.com/.

[19] Wikipedia contributors, *Large eddy simulation — Wikipedia, the free encyclopedia*, [Online; accessed 6-August-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Large_eddy_simulation&oldid=1233918184.

[20] gfortran developers, *Derived types and struct*, [Online; accessed 8-August-2024], 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/gfortran/Derived-Types-and-struct.html.

[21] Wikipedia contributors, *Green500 — Wikipedia, the free encyclopedia*, [Online; accessed 13-August-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Green500&oldid=1230059074.

Glossary

AST (Abstract Syntax Tree) tree structure that represents a program.. 5, 27

C-Struct class composed by only fundamental public members. It can holds pointers and static arrays but those should be at the end of the struct.. 32

FDS Fire Dynamic Simulator.. 2, 4, 5, 11, 18, 19, 22–26, 40–44

Hedgehog C++ library that uses data-flow graphs to run multithreaded code. . 2, 4–11, 13, 15–17, 19–28, 32, 40–42, 44

HMBE (HTGS Model-based engine), C++ library that uses data-flow graphs to run multithreaded code. It is the ancestor of Hedgehog.. 6

HPC High Performance Computing.. 2, 4, 11, 28, 43, 44

LLM (Large Language Model) AI model able to recognize and generate text. These models can process a large amount of data.. 26

parser the role of a parser is to analyse a string of symbols, conforming to the rules of a grammar. . 5, 27

SFINAE (Substitution Failure Is Not An Error), behavior of the compiler that does not generate an error when a template substitution fail.. 32, 33

Table of appendix

A Sierra specifications

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Address sizes:                   52 bits physical, 57 bits virtual
Byte Order:                      Little Endian
CPU(s):                          384
On-line CPU(s) list:             0-383
Vendor ID:                       AuthenticAMD
Model name:                      AMD EPYC 9654 96-Core Processor
CPU family:                      25
Model:                           17
Thread(s) per core:              2
Core(s) per socket:              96
Socket(s):                       2
Stepping:                        1
Frequency boost:                 enabled
CPU max MHz:                     3707.8120
CPU min MHz:                     1500.0000
Virtualization:                  AMD-V
L1d cache:                       6 MiB (192 instances)
L1i cache:                       6 MiB (192 instances)
L2 cache:                        192 MiB (192 instances)
L3 cache:                        768 MiB (24 instances)
NUMA node(s):                    2
NUMA node0 CPU(s):               0-95,192-287
NUMA node1 CPU(s):               96-191,288-383
```