

Université de Versailles Saint-Quentin-En-Yvelines
Master 2 Informatique Haute Performance et
Simulation



Module : Génie Logiciel pour le Calcul Scientifique

Rapport de projet

PAR : Nawal CHIBANE
Khadimou DIOP

Encadré par: MR. JULIEN BIGOT

28 janvier 2021

Table des matières

Introduction	3
1 Description du code	7
1.1 Une vue globale	7
1.2 La fonction Distributed2DField : :sync_ghosts	8
1.3 La classe CartesianDistribution2D	10
2 Ajout de fonctionnalités au code	11
2.1 Modification du système d'écriture des données	11
2.2 Modification du système de configuration	11
2.2.1 Définition du besoin	11
2.2.2 Conception	12
2.2.3 Réalisation	13
2.2.4 Post traitement des données	17
Conclusion	17

Table des figures

1.1	Graphe de dépendances	7
1.2	Graphe des appels du main	8
1.3	Graphe des appels de la fonction sync_gosts	9
1.4	La classe CartesianDistribution2D	10
2.1	Classe CmdParse et le graphe de dépendances	16
2.2	Ensemble des options du programme	17
2.3	La classe DataAvg	18
2.4	Graphe de dépendances de la classe DataAvg	18

Introduction

Le projet Heat Solver¹ s'agit d'une simulation de la propagation de la température dans un espace cartésien 2D, il met en oeuvre un solveur de l'équation de chaleur discrétisée avec la méthode des différences finies. Dans ce rapport, dans un premier temps, nous allons expliquer le fonctionnement du code fourni en mettant l'accent sur des fonctions et des classes spécifiques. Le code fourni n'est pas parfait et contient des imperfections, que se soit en termes d'expérience utilisateur (difficulté de gérer les paramètres en ligne de commande, visualisation des résultats sur la console ...), ou de performances (rapporter les résultats de toutes les itérations, ce qui dégrade les performances pour de longues simulations). Nous allons remédier à ces problèmes avec notre contribution, dans un deuxième temps, nous allons présenter les modifications que nous avons effectuées dans le code et les démarches suivies. Ces modifications couvrent trois parties :

- Modification du système d'écriture des données².
- Modification du système de configuration³.
- Post-traitement en ligne des données.

1. <https://github.com/jbigot/Projet-GLCS-2020-2021>.

2. Faite par le membre : Khadimou DIOP

3. Faite par le membre : Nawal CHIBANE

Description du code

1.1 Une vue globale

Le main du programme se trouve dans le fichier **simpleheat.cpp**, récupère les paramètres depuis la ligne de commande, fait les instantiations nécessaires et lance la simulation. Aussi, après les initialisations nécessaires dans la fonction **run()** de la classe **Simulation**, cette fonction, dans une boucle d'itérations, exécute la fonction **iter()** qui se trouve dans la classe **FinitediffHeatSolver**.

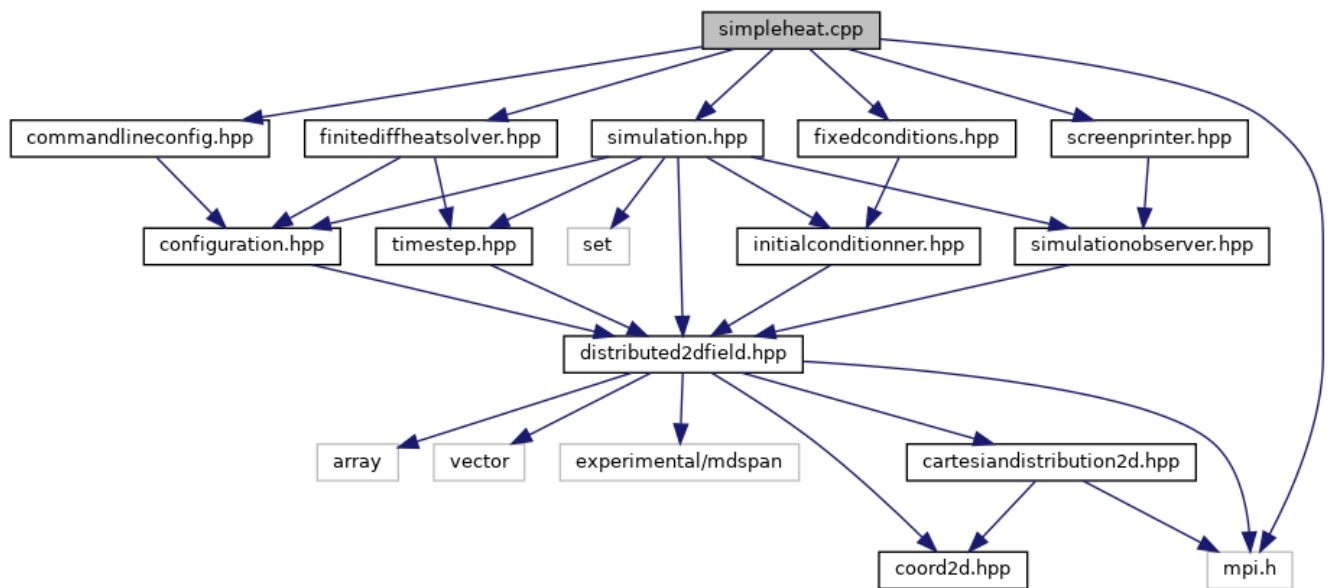


FIGURE 1.1 – Graphe de dépendances

Dans la classe **FinitediffHeatSolver** la fonction **iter** permet de calculer la chaleur au moment $(t+\text{pas})$ pour chaque élément d'un bloc de données courant **Distributed2DField cur** à l'instant t , et stocke les nouvelles valeurs dans un nouveau bloc **Distributed2DField next**. En effet chaque processus MPI s'occupe d'un bloc, et pour le calcul des nouvelles valeurs, a besoin d'accéder aux données des zones fantômes de son bloc, celles ci mises à jour par les processus voisins.

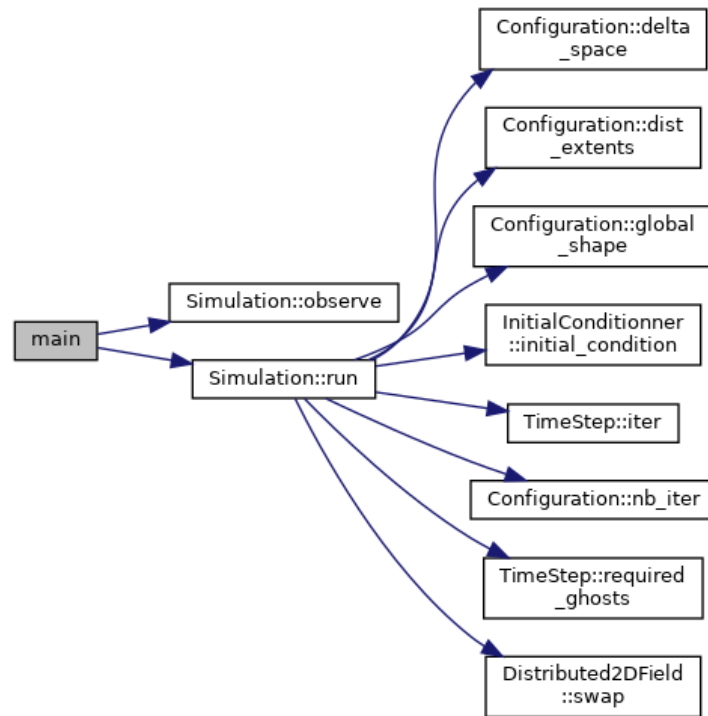
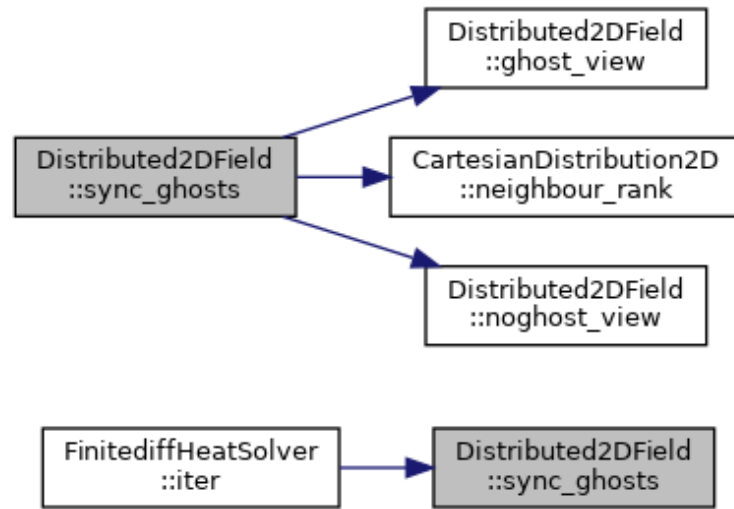


FIGURE 1.2 – Graphe des appels du main

1.2 La fonction `Distributed2DField : :sync_ghosts`

La fonction `Distributed2DField : :sync_ghosts` fait des échanges entre processus voisins dans chaque direction. Elle permet à chaque process de mettre à jour les zones fantômes des blocs voisins selon les directions (up, down, left, right). Pour cela, pour le processus MPI actuel, la fonction `sync_gosts` récupère le rang des processus voisins (up, down, left, right) en appelant la fonction `neighbour_rank(direction)` implémentée dans la classe `CartesianDistribution2D`. En effet, chaque instance de la classe `Distributed2DField` possède un objet du type `CartesianDistribution2D` qui décrit la distribution des données par rapport aux processus MPI. Une fois que les voisins sont identifiés, le processus actuel communique avec chacun d’eux et lui envoie le message qui contient la ligne/colonne qui sert de zone fantôme au voisin concerné avec la fonction MPI avec la fonction de MPI `MPI_Sendrecv`.

Ici on commence par la direction DY. On récupère les rangs des processus voisins en dessous et au dessus des domaines locaux. Puis on stocke la coordonnée du dernier bloc de données de la même taille que la zone fantôme dans la zone non fantôme. Par la suite en utilisant une distribution cartésienne, on envoie la première ligne de blocs de la zone fantôme en dessous dans la zone non fantôme puis on reçoit la ligne de la zone fantôme d’au dessus avec `MPI_Sendrecv()`. En suite on envoie la dernière ligne de blocs au dessus de la zone fantôme et on reçoit la zone fantôme d’en dessous avec `MPI_Sendrecv()` en utilisant une distribution cartésienne. Enfin on répète le même processus pour la di-

FIGURE 1.3 – Graphe des appels de la fonction `sync_ghosts`

rection DX en remplaçant donc le UP et le DOWN par respectivement RIGHT et LEFT. On envoie de la gauche et reçoit de la droite et vice versa.

La fonction **MPI_Sendrecv** se définit comme suit :

```

1 int MPI_Sendrecv(const void *sendbuf, int sendcount,
2                 MPI_Datatype sendtype, int dest, int sendtag,
3                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
4                 int source, int recvtag, MPI_Comm comm, MPI_Status * status)

```

Dans notre cas :

sendbuf :noghost_view : l'adresse de la ligne ou colonne à envoyer (vecteur :ligne/col)

sendcount : nombre d'éléments à envoyer = 1 dans notre cas

sendtype : m_ghost_row ou m_ghost_col

dest : rang du processus voisin

sendtag : COMM_TAG

recvbuf : ghost_view(direction) : l'adresse de la zone fantôme réceptrice

recvcount : nombre d'éléments à recevoir

recvtype : m_ghost_row ou m_ghost_col

source : rang du processus actuel

recvtag : COMM_TAG

comm : MPI_Comm dans la "classe CartesianDistribution2D"

1.3 La classe CartesianDistribution2D

En effet, comme déjà mentionné, un objet **CartesianDistribution2D** est un descripteur de la distribution des données sur les processus MPI, une instance est créée pour chaque processus lors de l'appel au constructeur **Distributed2DField** lors de l'initialisation des données. Son constructeur prend en entrée le communicateur MPI et un objet 2D décrivant le nombre de processus MPI par ligne et pas colonne et crée un communicateur cartésien auquel les informations de topologie cartésienne sont attachées avec la fonction **MPI_Cart_create**, ie, le nombre de dimensions d'un bloc de données = 2 dimensions et le nombre de processus MPI dans chaque dimension.

CartesianDistribution2D Class Reference

```
#include <cartesiandistribution2d.hpp>
```

Public Member Functions

CartesianDistribution2D (MPI_Comm comm, Shape2D shape)	
Shape2D	extents () const
int	extent (Dimension2D dim) const
Coord2D	coord () const
int	coord (Dimension2D dim) const
const MPI_Comm	communicator () const
MPI_Comm	communicator ()
int	neighbour_rank (Direction2D direction)
int	size () const
int	rank () const

FIGURE 1.4 – La classe CartesianDistribution2D

MPI fournit une topologies cartésiennes permettant de décrire des découpages de grilles en 2D dans notre cas, ce découpage permet de :

Obtenir le rang du processus à partir de ses coordonnées

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)

Obtenir les coordonnées du processus à partir de son rang

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)

Récupérer les informations de topologie

MPI_Cart_get(MPI_Comm comm, int maxdims, int dims[], int periods[], int coords[])

cette fonction est utilisée dans la classe **CartesianDistribution2D** par les fonctions **coord** et **extents** Renvoie les rangs des voisins, en fonction d'une direction et d'un décalage

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST)

Cette dernière à été utilisée dans la classe **CartesianDistribution2D** par la fonction **neighbour_rank** afin de calculer les rang des quatre voisins en variant la direction de (DX et DY) et le décalage DISP de (-1 et 1).

Ajout de fonctionnalités au code

2.1 Modification du système d'écriture des données

J'ai écrit une nouvelle classe qui stocke les données au format HDF5 dans un dossier que j'ai appelé **writing_datahdf5**. Cette classe est inspirée de la classe **ScreenPrinter** à la différence près qu'on rajoute la bibliothèque `<hdf5.h>` grâce à laquelle on pourra écrire les données dans le format hdf5. Pour ce faire, on crée un dataspace qui contient les dimensions de la grille et leurs tailles actuelles. Ensuite on crée le dataset pour fixer les propriétés de notre jeux de données en utilisant le dataspace défini précédemment. Puis on utilise un simple hyperslab pour choisir l'endroit où l'on veut écrire dans le fichier dataspace. Enfin pour écrire dans la mémoire dataspace j'alloue une mémoire suffisante pour stocker les éléments de données à transférer en l'initialisant avec les données à écrire. Par ailleurs, on se place en mode data transfert pour contrôler les opérations I/O lors du transfert. Par la suite on a ajouté une fonction dans la classe Configuration pour gérer la fréquence des itérations de manière que cette fréquence soit configurable par l'utilisateur.

2.2 Modification du système de configuration

2.2.1 Définition du besoin

La simulation étudiée nécessite un ensemble de paramètres en entrée, il s'agit essentiellement de :

- Le nombre d'itérations à exécuter.
- Les dimensions 2D de l'espace objet de simulation.
- La distribution 2D des processus MPI sur cet espace.
- La valeur δt : pas de discrétisation temporel.
- La valeur δx : pas de discrétisation spatial par rapport à la première dimension.
- La valeur δy : pas de discrétisation spatial par rapport à la deuxième dimension.

Nous rappelons que l'implémentation donnée récupère ces paramètres depuis la ligne de commande comme suit :

```
1 $ mpirun -n 2 ./simpleheat 10 4 8 1 2 0.125 1 1
```

Cette manière de faire présente plusieurs inconvénients :

- Nous n'avons pas d'information sur l'identité des paramètres.
- Obligation de respecter l'ordre défini.
- Difficulté de gérer un grand nombre de paramètres.
- On peut se perdre facilement si on oublie un paramètre et chercher sa place.
- Pas de gestion de paramètres par défaut.
- Mauvaise UX.

Le but de cette partie est de proposer une solution aux problèmes cités ci-dessous, nous allons détailler dans ce qui suit notre approche.

2.2.2 Conception

Nommage des paramètres :

Nous allons en premier temps donner une identité à nos paramètres dans la ligne de commande comme suit :

- **nb_iter** : le nombre d'itérations
- **height** : longueur de l'espace de données global.
- **width** : largeur de l'espace de données global.
- **process_height** : la distribution des données selon la largeur.
- **process_width** : la distribution des données selon la longueur.
- **delta_t** : pas de discrétisation temporel.
- **delta_x** : pas de discrétisation spatial selon la longueur.
- **delta_y** : pas de discrétisation spatial selon la largeur.
- **file** : le fichier de configuration.
- **freq** : fréquence d'écriture des données.

Dans la ligne de commande, nous allons introduire chaque paramètre en indiquant son identifiant précédé par "-" suivi par = sa valeur, ainsi la commande précédente devient :

```
1 mpirun -n 2 ./simpleheat --nb_iter=10 --height=4 --width=8
2 --process_height=1 --process_width=2 --delta_t=0.125 --delta_x=1
3 --delta_y=1
```

Paramètres par défaut :

Nous allons fournir à l'utilisateur un fichier de configuration default.ini qui va contenir les valeurs par défaut des paramètres. Si l'utilisateur n'introduit pas en ligne de commande les paramètres ou bien il introduit que certains d'entre eux, les paramètres manquants vont avoir les valeurs du fichier.

```
1 mpirun -n 2 ./simpleheat
```

Choix d'un fichier de configuration :

Nous allons fournir à l'utilisateur l'option de choisir un fichier de configuration qui va contenir les valeurs des paramètres personnalisés. Il pourra l'introduire au programme

avec l'option "file" en indiquant dans la ligne de commande le chemin du fichier comme valeur à l'option `-file`, exemple :

```
1 mpirun -n 2 ./simpleheat --file=config.ini
```

Si l'utilisateur n'introduit pas dans son fichier tout les paramètres, les valeurs manquantes vont être complétée par leurs options en ligne de commande si elles sont présente, sinon par les valeurs par défaut dans le fichier `default.ini`.

2.2.3 Réalisation

Outils utilisés :

Il existe plusieurs librairies qui peuvent répondre à notre besoin, nous citons Boost, Clara, CLI11, The Lean Mean C++, getopt, Ktopt, args ...

Pour implémenter la solution proposée, nous avons utilisé l'API QT5. La classe **QCommandLineParser** pour analyser les options en ligne de commande et la classe **QSettings** pour gérer les fichiers de configuration.

Qt5 est un framework initialement prévu pour faciliter la création d'interfaces graphiques pour le langage de programmation C++. Au fil du temps et des nouveaux apports aux bibliothèques de Qt, ce framework s'est étendu progressivement au delà des interfaces graphiques pour fournir une bibliothèque de composants facilitant l'utilisation du langage C++ (sockets, fichiers, structure de données, threads, synchronisation, ...), allant jusqu'à permettre la portabilité du code source à différentes plateformes. La portabilité des applications n'utilisant que des composants Qt se fait par simple recompilation du code source. Les environnements supportés sont les Unix (dont Linux) qui utilisent le système graphique X Window System, Windows et Mac OS X.¹

Qt support depuis sa version 5 l'analyse de la ligne de commande avec un niveau d'abstraction élevé, les deux classes **QCommandLineParser** et **QSettings** sont très simples et faciles à utiliser, et répondent à notre besoin.

Ajour d'une bibliothèque CLIPARSER :

Nous avons implémenté une bibliothèque indépendante que nous avons appelé **CLIPARSER** qui contient une classe **CmdParse**, celle ci implémente la classe **Configuration** et redéfinit les fonctions virtuelles de cette dernière. Ceci dans le but de pas pas effectuer beaucoup de changement dans le code.

La classe CmdParse :

Pour éviter quelques erreurs dans le programme, nous avons initialisé des constantes qui représentent les valeurs pas défauts des paramètres.

```
1 // the default values
```

1. https://fr.wikibooks.org/wiki/Programmation_Qt/Introduction

```

2 static int DEFAULT_ITER = 0;
3 static int DEFAULT_HEIGHT = 0;
4 static int DEFAULT_WIDTH = 0;
5 static int DEFAULT_PROC_HEIGHT = 0;
6 static int DEFAULT_PROC_WIDTH = 0;
7 static int DEFAULT_DELTA_X = 0;
8 static int DEFAULT_DELTA_Y = 0;
9 static QString DEFAULT_DELTA_T = "0";
10 static QString DEFAULT_CONFIG_FILE = "../default.ini";
11 static int DEFAULT_WRITE_FREQ = 0;

```

CmdParse dans son constructeur, grâce à la classe **QSettings** lit le fichier **default.ini** s'il existe, et récupère les valeurs et met à jour ces constantes.

QVariant **QSettings::value(const QString &key, const QVariant &defaultValue = QVariant()) const**

```

1 // Update the default values with those in the default config file if it
  exists
2 if (QFile(DEFAULT_CONFIG_FILE).exists())
3 {
4     QSettings settings(DEFAULT_CONFIG_FILE, QSettings::IniFormat);
5     // retrieve the setting
6     DEFAULT_DELTA_T = settings.value("params/delta_t", DEFAULT_DELTA_T).
      toString();
7     DEFAULT_DELTA_X = settings.value("params/delta_x", DEFAULT_DELTA_X).
      toInt();
8     DEFAULT_DELTA_Y = settings.value("params/delta_y", DEFAULT_DELTA_Y).
      toInt();
9     DEFAULT_ITER = settings.value("params/nb_iter", DEFAULT_ITER).toInt();
10    DEFAULT_HEIGHT = settings.value("params/height", DEFAULT_HEIGHT).toInt
      ();
11    DEFAULT_WIDTH = settings.value("params/width", DEFAULT_WIDTH).toInt();
12    DEFAULT_PROC_HEIGHT = settings.value("params/process_height",
      DEFAULT_PROC_HEIGHT).toInt();
13    DEFAULT_PROC_WIDTH = settings.value("params/process_width",
      DEFAULT_PROC_WIDTH).toInt();
14    DEFAULT_WRITE_FREQ = settings.value("params/freq", DEFAULT_WRITE_FREQ)
      .toInt();
15 }

```

La classe **QCommandLineOption** permet de rajouter une option avec une valeur par défaut, puis chaque option est rajoutée au parseur instance de **QCommandLineParser**.

```

1 // Add config file option (--file)
2 QCommandLineOption fileOption("file",
3     QCoreApplication::translate("simpleheat", "config file name"),

```

```
QCommandLineOption::QCommandLineOption(const QStringList &names, const QString
&description, const QString &valueName = QString(), const QString &defaultValue = QString())
```

```
bool QCommandLineParser::addOption(const QCommandLineOption &option)
```

```
4     QCoreApplication::translate("simpleheat", "file name"),
5     std::to_string(DEFAULT_DELTA_Y).c_str());
6     parser.addOption(fileOption);

1 // Add config file option (--file)
2     QCommandLineOption fileOption("file",
3     QCoreApplication::translate("simpleheat", "config file name"),
4     QCoreApplication::translate("simpleheat", "file name"),
5     std::to_string(DEFAULT_DELTA_Y).c_str());
6     parser.addOption(fileOption);
```

Enfin, dans le cas où l'utilisateur introduit un fichier de configuration, nous lisons les valeurs depuis celui-ci et mettons à jour les variables locales de la classe, sinon à partir des options.

```
1 // if the user introduces a config file with the option --file
2     if (parser.isSet(fileOption))
3     {
4         QString file_name=parser.value(fileOption);
5         QSettings settings(file_name, QSettings::IniFormat);
6         // get the values from the chosen config file if it exist
7         // if one value miss in the file, get it from the default values
8         if (QFile(file_name).exists())
9         {
10             m_nb_iter = (parser.isSet(nbIterOption)) ? parser.value(
nbIterOption).toDouble() : settings.value("params/nb_iter",
DEFAULT_ITER).toDouble();
11             .
12             .
13             .
14         }
15         else
16         {
17             cerr << "ERROR: file " << file_name.toStdString().c_str() << "
doesn't exist! \n" << endl;
18             exit(1);
19         }
20     }
21     else
22     {
23         // get the values from their options
24         // if one option miss, the default value will be returned
25         m_nb_iter = parser.value(nbIterOption).toInt();
```

26
27
28
29

```

        .
        .
        .
    }

```

Nous présentons dans la figure 2.1 un résumé de la classe implémentée ainsi que le graphe de dépendances généré par doxygen.

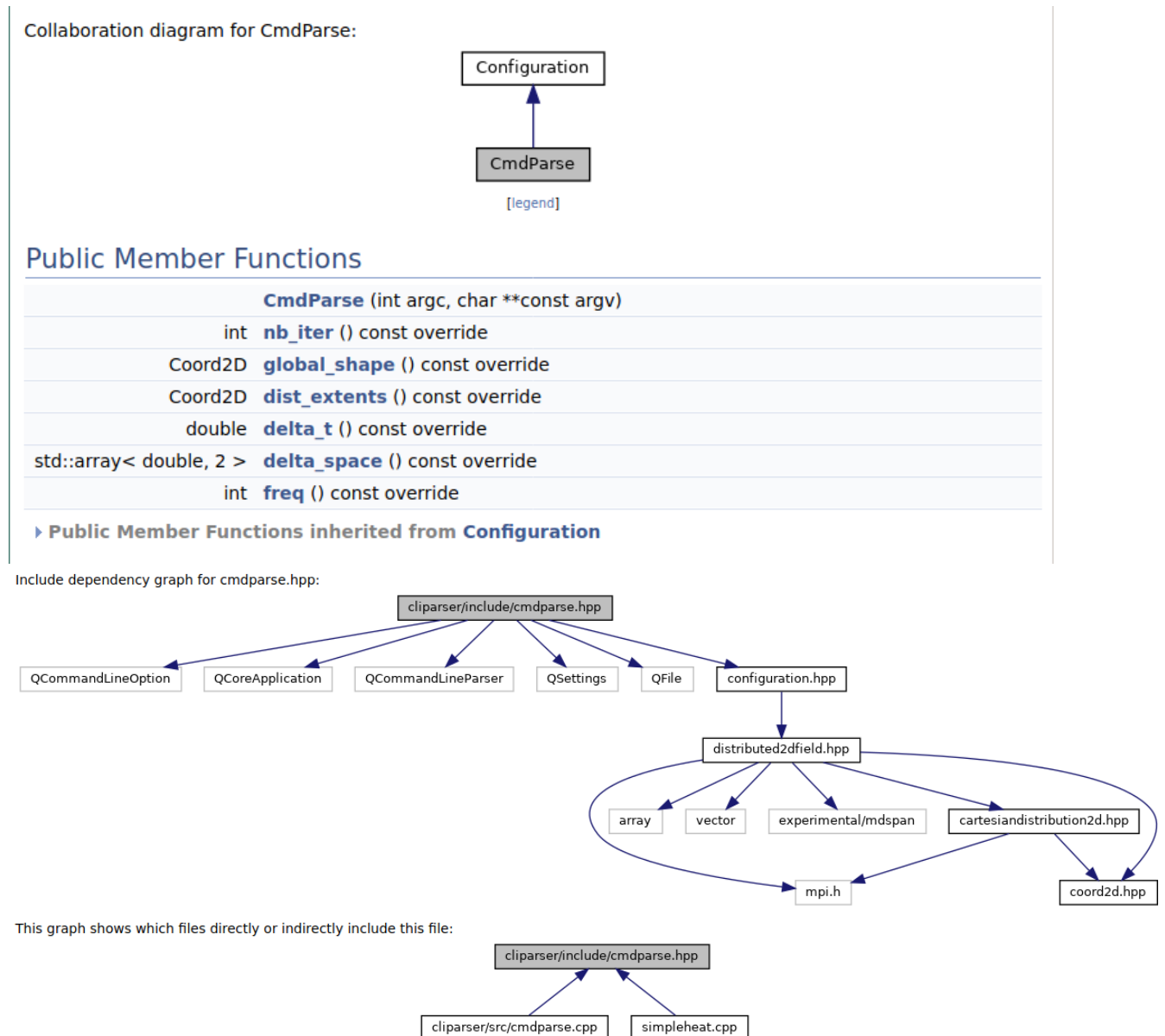


FIGURE 2.1 – Classe CmdParse et le graphe de dépendances

Enfin, nous nous montrons dans la capture 2.2 le résultat de la commande `--help`

```

Projet_NK/ProjetGLCS/Projet-GLCS-2020-2021/build$ ./simpleheat --help
Usage: ./simpleheat [options]
This is a simple heat solver

Options:
  -h, --help                Displays this help.
  -v, --version             Displays version information.
  --file <file name>       config file name
  --nb_iter <nb_iter>      number of iterations to execute
  --height <height>        height of the global data field
  --width <width>          width of the global data field
  --process_height <process_height> height of the data distribution
  --process_width <process_width> width of the data distribution
  --delta_t <delta_t>      time difference between two consecutive
                           points
  --delta_x <delta_x>      width difference between two consecutive
                           points
  --delta_y <delta_y>      height difference between two consecutive
                           points
  --freq <frequence>       frequence of writting to the hdf5 file
(base) noelle@noelle-HP-EliteBook-Folio-1040-G1:~/Desktop/S2/GLCS/Projet_glcs/Pr

```

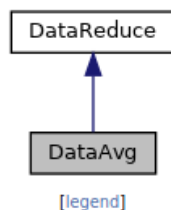
FIGURE 2.2 – Ensemble des options du programme

2.2.4 Post traitement des données

Nous avons crée une nouvelle bibliothèque pour appelée **posttreat** pour le post-traitement des données, cette bibliothèque contient une classe **DataAvg** qui implémente une interface **DataReduce** et redéfinit ses fonctions. La moyenne de l'ensemble des données d'un bloc est calculée lors de l'appel au constructeur, la classe contient une variable **m_data_avg** qui contient la moyenne calculée, sa valeur est retournée en appelant le getter **average()**. Ce calcul de la moyenne peut être utilisé pour réduire les écritures dans le fichier hdf5, nous pouvons, en la comparant à un seuil (introduit par l'utilisateur comme paramètre) décider si les données méritent d'être écrites ou pas.

Les figures 2.3 et 2.4 montrent la classe rajoutée ainsi que ses dépendances.

Collaboration diagram for DataAvg:



Public Member Functions

DataAvg (const Distributed2DField &data)
double average () const override
▼ Public Member Functions inherited from DataReduce
virtual ~DataReduce ()=default
The destructor. More...

FIGURE 2.3 – La classe DataAvg

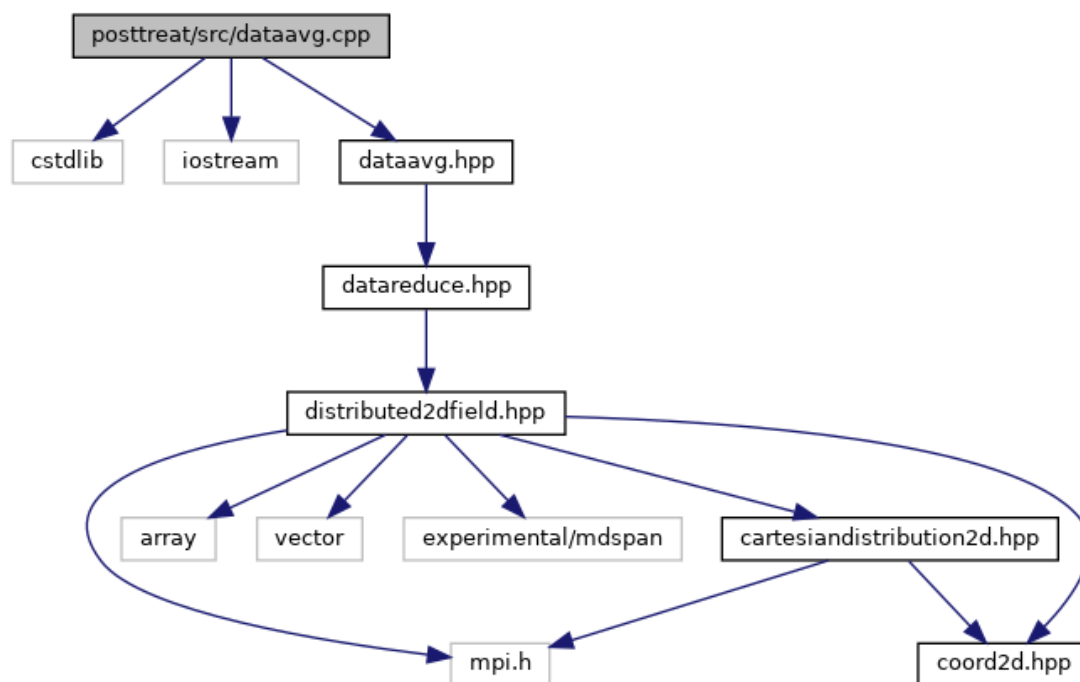


FIGURE 2.4 – Graphe de dépendances de la classe DataAvg

Conclusion

Nous avons, grâce à ce projet, pu voir un exemple de simulation dans un contexte scientifique ainsi que l'usage des bonnes pratiques du génie logiciel et des patterns de conception pour la réalisation de telles applications (Inversion de contrôle, pattern Observer, gestion automatique des versions avec git, génération automatique de la documentation avec Doxygen, utilisation de la technologie de conteneurisation docker ...). Cet exemple nous a permis d'avoir une vision sur le calcul scientifique parallèle, sur un cas plus au moins concret, nous avons manipulé deux bibliothèques indispensables qui sont **MPI** et **Hdf5**, et découvert d'autres durant le projet (Qt5).