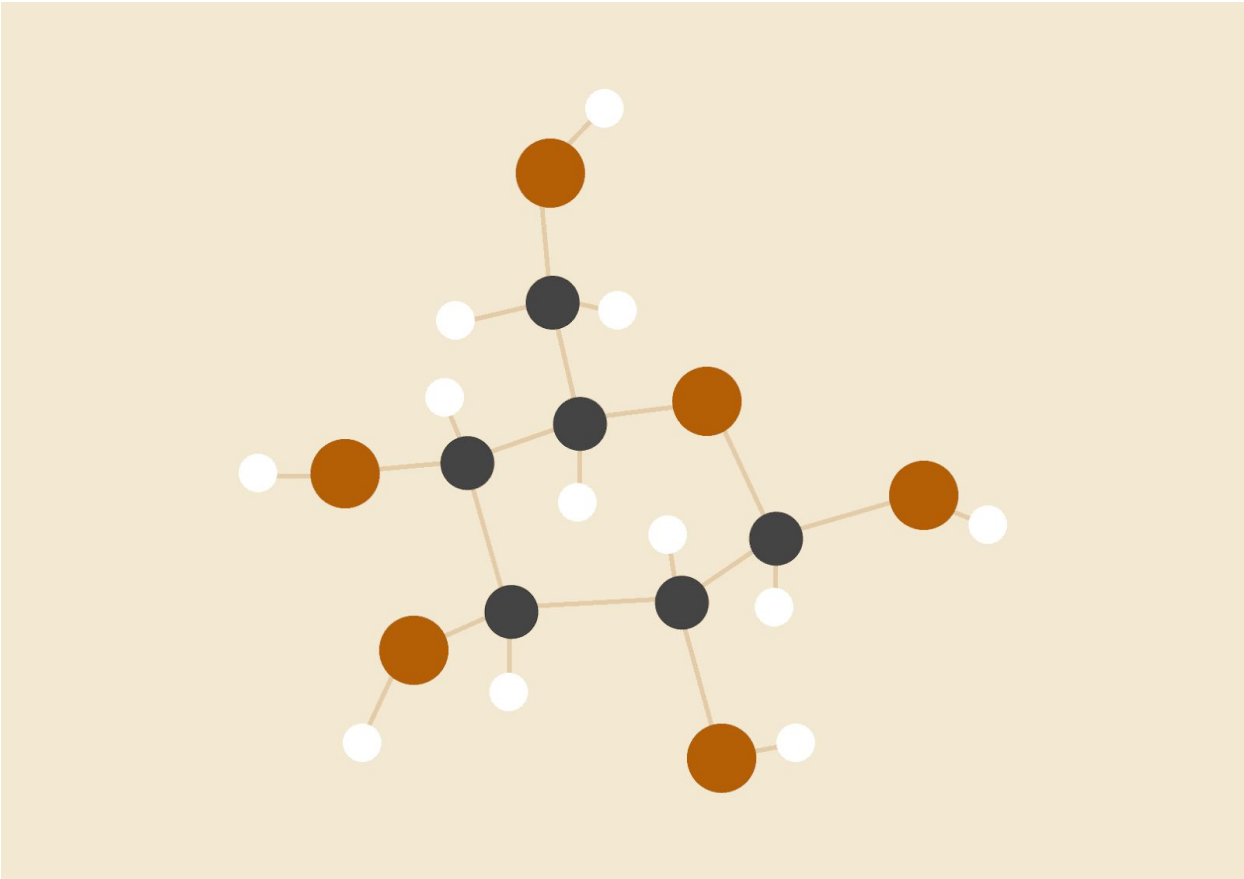
RAPPORT DU PROJET GLCS



AZZAZ Hichem |SMAIL Karim

28/01/2021

Introduction:

Dans notre projet, on va présenter un solveur C++ de l'équation de la chaleur.

Le projet est composé de 3 bibliothèques principales, qui représente les différentes fonctionnalités principales du projet à savoir l'affichage des données, le solveur utilisé dans ce cas les différents streamings, et le code afin de profiter de différentes fonctionnalités de C++ et la programmation orienté objet en générale.

Différentes fonctionnalités de C++ sont utilisées dans ce projet à savoir le design pattern et la version de contrôle .

Un CMakeList est utilisé pour le building et le linking des différentes bibliothèques.

Description du fonctionnement du code fourni:

Globalement,le projet met en œuvre un solveur de l'équation de la chaleur.

Plusieurs Classes et fonctions ont été implémentés dans ce projet, afin d'inplémenté le solveur de l'équation de la chaleur, la description de ces classes et fonctions est comme suit:

Dans la fonction FinitediffHeatSolver::iter, il est implémenté le code de l'équation de la chaleur (Ut+(i,j), avec Ut+ = next et Ut = cur), on parle de stencil car le calcul des valeurs à chaque coordonnée dépend des valeurs aux coordonnées voisines du pas de temps précédent.

condition aux limites:

Dans le code fourni, la fonction **FixedConditions::initial_condition** positionne les conditions suivantes :

— conditions initiales : U0 = 0 sur tout le domaine,

— conditions au limites en haut, en bas et à droite : conditions de Dirichlet, Ut(i,j) = 0,

— conditions au limites à gauche : conditions de Dirichlet, Ut(i,0)= 2097.152,

La matrice est divisée en domaine local(toute la matrice est considérée comme domaine global). Ces domaines communiquent entre eux à l'aide d'MPI.

Il existe plusieurs distribution de domaine de la matrice(celle au limite n'est pas la même que celle du milieu, car pour calculer chaque élément, on a pas besoin de la valeur de son voisin, qui peut être dans une autre zone)

Cette distribution de domaine est implémentée par les classes Distributed2DField qui représente un bloc local de données et CartesianDistribution2D qui identifie la distribution du domaine local au sein du domaine global. La classe Distributed2DField expose des “vues” (view) sous la forme de fonctions membres :

— full_view donne accès aux données avec un système de coordonnées qui commence à 0 au début des zones fantômes,

— noghost_view donne accès aux données avec un système de coordonnées qui commence à 0 après les zones fantômes et n’expose que le domaine local,

— ghost_view donne accès aux données avec un système de coordonnées qui commence à 0 au début de chaque zone fantôme et donne accès à cette zone fantôme spécifique.

La fonction sync_ghosts synchronise les zones fantômes entre processus voisins.

Description du code:

1. **simpleheat.cpp: (le main)**

construire la ligne de commande avec ses arguments, en utilisant la fonction config() de la classe CommandLineConfig.

Construire l'équation de la chaleur Ut+(i,j). en utilisant la fonction heat_solver(config) de la classe FinitediffHeatSolver.

On fixe les conditions initiales avec la fonction init de la classe FixedConditions.

Construire le simulateur en précisant le rang du processus MPI, la configuration, l'équation de la chaleur et les conditions initiales.

2. **CommandLineConfig.cpp: (la classe de configuration)**

Contient les arguments de configuration du programme:

- Le nombre d'itération à exécuter n_iter

- La forme du champ de données (2D) dans notre cas m_global_shape
- Le forme de la distribution des données m_global_shape
- Le temps d'execution entre deux point consecutifs m_delta_t
- La différence d'espace entre deux points consécutifs m_delta_space

Le constructeur de cette classe, permet construire l'analyseur d'arguments de ligne de commande et retourne erreur en cas d'erreur dans les arguments.

3. **FinitediffHeatSolver.cpp : la classe qui construit le solveur de l'équation de la chaleur**

La fonction iter(), prend en paramètre le coordonné courant (cur) et le coordonné suivant (next) à calculer.

Elle permet de fixer les champs de la zone fantôme de l'élément next.

Une implémentation de 'TimeStep' qui implémente un solveur d'équation de chaleur via la méthode des différences finies sur une grille cartésienne.calcule de temps d'exécution entre deux points consécutifs m_delta_t.

4. **FixedConditions.cpp : la classe qui fixe les conditions initiales**

Le constructeur de cette classe permet de: définir les conditions initiale qui définit simplement l'initiale prédéfinie et les conditions aux limites de Dirichlet (définie dans la description du projet):

- A t = 0, 0 est mis sur tout le domaine,
- A la limite en haut, à droite et en bas, la valeur 0 est également définie
- A la limite à gauche, 2097.152 est utilisé.

5. **Simulation.cpp : La classe de simulation principale qui implémente le pas de temps.**

Cette classe a besoin des services des autres classes pour fournir la plupart de son comportement.La classe a besoin de :

- Une instance de Configuration fournit les paramètres spécifiés par l'utilisateur,
- Une instance InitialConditionner fournit les conditions initiales et limites tandis qu'une instance TimeStep implémente l'opérateur à appliquer à chaque pas de temps.

6. **La fonction run() permet d'exécuter le simulateur.**

La simulation est observable par les instances de SimulationObserver grâce aux fonctionsobserve() et unobserve().

Modification du système d'écriture des données:

La bibliothèque utilisée pour l'écriture des données sur un fichier HDF5 a été implémenté en modifiant PrintScreen déjà existante:

La méthode de d'écriture est faite en modifiant quelques parties de la méthode d'affichage, en suivant ces étapes :

1. création du fichier HDF5 par le processus racine comme suite :

```
   if ( data.distribution().rank() == 0 ) {

       // Create an HDF5 file to store the varaiable

       //1) create a file named : file.h5 (HDF5 extention)

       hid_t file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC,H5P_DEFAULT,
H5P_DEFAULT);

       //2) set the dimention of the matrix to store inside the

       hsize_t dims[2] = {data.distribution().extent( DX
),data.noghost_view().extent( DY )};

       // create the dataspace associated to the simulationand create the
dataset

       hid_t dataspace_id = H5Screate_simple(2, dims, NULL);

       dataset_id = H5Dcreate(file_id, "S", H5T_STD_I32BE,dataspace_id,
H5P_DEFAULT,H5P_DEFAULT, H5P_DEFAULT);

   }
```

L'identifiant du dataset sera communiqué aux autres processus en utilisant le broadcast du MPI:

```
MPI_Bcast(&dataset_id, 1,H5T_STD_I32LE, 0,
data.distribution().communicator());
```

2. Préparation de l'espace de transfert vers le dataset, par chacun des processus :

```
   // preparing the simple for transferting the data to the dataset

   hsize_t dims[1] ;

   hid_t mspace_id = H5Screate_simple(1, dims, NULL);

   // Get the space id for our dataset of the HDF5 file

   hid_t fspace_id = H5Dget_space(dataset_id);

   // The coords will be determined for every element of the simulation
data

   hsize_t start[2];

   // We write a single element at the time

   hsize_t count[2] = {1, 1};

   // the array containing the value to write in the dataset

   double value[1];
```

3. En utilise les même boucles imbriquées de la fonction d'affichage pour avoir les élément de la matrice de simulation, et au lieu de les afficher, en les transfert au dataset, comme suite :

```
 if ( 0 == data.noghost_view(yy, xx) ) {

      value[0] = 0;

 } else {

   value[0] = data.noghost_view(yy, xx);
```

```
 }

// insert the element inside the

H5Sselect_hyperslab(fspace_id, H5S_SELECT_SET, start, NULL, count,NULL);

H5Dwrite(dataset_id, H5T_NATIVE_INT, mspace_id, fspace_id,H5P_DEFAULT,
value);
```

Les coordonnées du start sont modifier à chaque itération :

```
start[1] = yy;

start[0] = xx;
```

A la fin, le processus racine , ferme le dataset, l'espace data , ainsi que le le fichier HDF5:

```
   if ( data.distribution().rank() == 0 ) {

       H5Dclose(dataset_id);

       H5Sclose(dataspace_id);

       H5Fclose(file_id);

   }
```

Modification du système de configuration

Dans ce projet, on a utilisé la librairie Boost.ProgramOptionde du C++, pour mettre un en place un système de configuration des paramètres du programme.

Boost.ProgramOptions est une bibliothèque qui facilite l'analyse des options de ligne de commande. Si vous développez des applications avec une interface utilisateur graphique, les options de ligne de commande ne sont généralement pas importantes.

Pour analyser les options de ligne de commande avec Boost.ProgramOptions, on a trois étapes à suivre:

Définissez les options de ligne de commande. Vous leur donnez des noms et spécifiez ceux qui peuvent être définis sur une valeur. Si une option de ligne de commande est analysée comme une paire clé / valeur, vous définissez également le type de la valeur, par exemple, s'il s'agit d'une chaîne ou d'un nombre.

Utilisez un analyseur pour évaluer la ligne de commande. Vous obtenez la ligne de commande à partir des deux paramètres de main(), qui sont généralement appelés argc et argv .

Stockez les options de ligne de commande évaluées par l'analyseur. Boost.ProgramOptions propose une classe dérivée de std::map qui enregistre les options de ligne de commande sous forme de paires nom / valeur. Ensuite, vous pouvez vérifier quelles options ont été stockées et quelles sont leurs valeurs.

Dans notre cas, les paramètres utilisés pour la simulation sont peu lisibles, leurs lecteurs de la ligne de commande est longue et l'ajout de nouveaux paramètres est rendu complexe par l'impossibilité d'avoir des valeurs par défaut. Donc pour rendre les paramètres du programme lisible, compréhensible, et pour avoir des valeurs par défaut de ces paramètres,

On veut remplacer la classe commandeLineConfig qui analyse les options passées sur la ligne de commande, par une implémentation basée sur la bibliothèque Boost.Ptogram_option déjà défini.

L'installation de la bibliothèque Boost.ProgramOption :

On a rajouté dans le projet, un répertoire nommé "boost", qui contient les fichiers sources, qui définissent les fonctionnalités proposées dans cette bibliothèque, et qui nous permet de manipuler les fonctions de contrôle de flux d'entrée de programme.

Un CMakeLists est mis en place pour relier cette bibliothèque au autres bibliotheque du programme :

```
add_library(boost
        src/cmdline.cpp
        src/config_file.cpp
        src/convert.cpp
        src/options_description.cpp
```

```
      src/parsers.cpp

      src/positional_options.cpp

      src/split.cpp

      src/utf8_codecvt_facet.cpp

      src/value_semantic.cpp

      src/variables_map.cpp

      src/winmain.cpp)
```

```
target_include_directories(boost PUBLIC include/)

target_link_libraries(boost PUBLIC GLCS2020_project::baselib)

add_library(GLCS2020_project::boost ALIAS boost)
```

Une modification aussi et obligatoire dans le CMakeLists du programme principale, en ajoutant la sub_dirictory nommé "boost" pour faire appel à ce CMake lors de la compilation du programme et la génération du makefile.

Les paramètres du programme et leur implémentation avec Boost.program_option:

On a créé chaque variable du programme, en lui attribuant une valeur par défaut:

Dans le programme principale:

```
("Nb_iter", po::value<int>()->default_value(10), "Nb_iter")

("height", po::value<int>()->default_value(4), "height")

("width", po::value<int>()->default_value(8), "width")

("process_height", po::value<int>()->default_value(1), "process_height")

("process_width", po::value<int>()->default_value(1), "process_width")

("delta_t", po::value<float>()->default_value(0.125), "delta_t")
```

```
("delta_x", po::value<float>()->default_value(1), "delta_x")

("delta_y", po::value<float>()->default_value(1), "delta_y")

("file_name", po::value<char*>()->default_value(NULL), "file_name")
```

Toutes ces variables seront stockées dans une classe dérivée de std::map (dans notre cas vm) qui enregistre les options de ligne de commande sous forme de paires nom / valeur. (au cas où une valeur manquante, la valeur par défaut sera utilisée).

```
      po::variables_map vm;

      po::store(parse_command_line(argc, argv, desc), vm);

      po::notify(vm);
```

Pour préparer l'étape de constriction du le solveur d'équation de chaleur, il définir le paramètre config utilisé dans la FinitediffHeatSolver, pour cela on a créé un vecteur qui contient les valeurs des paramètres du programme, et on l'a utilisé pour créé une instance config de la classe configuration:

```
    int N = 8;

    char* param = (char*)malloc(sizeof(char)*N);

    param[0] = vm.count("Nb_iter");

    param[1] = vm.count("height");

    param[2] = vm.count("width");

    param[3] = vm.count("process_height");

    param[4] = vm.count("process_width");

    param[5] = vm.count("delta_t");

    param[6] = vm.count("delta_x");

    param[7] = vm.count("delta_y");

  // Construct the command-line arguments parser

    Config config(N, param);
```

Conclusion :

Pour conclure, dans ce projet on a fait une description du fonctionnement du code donné. Ensuite, on a fourni une nouvelle classe pour remplacer l'écriture des données sur la console qui stocke les données au format HDF5. A la fin on a proposer une nouvelle implémentation qui remplace la classe "CommandLineConfig" basé sur la librairie Boost.Ptogram_option qui analyse les options passées sur la ligne de commande, une implémentation qui supporte des paramètres par défaut et qui permet d'avoir plus de lisibilité.

Merci.