UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Parallélisation d'un outil de simulation hydraulique numérique

Mémoire présenté par
Emilio GAMBA , Jean-Baptiste MACQ

en vue de l'obtention du grade de Master
Ingénieur Civil en Informatique

Promoteur(s)
Sandra SOARES-FRAZÃO, Etienne RIVIÈRE

Lecteur(s)
Damien FRANÇOIS, Jonathan LAMBRECHTS

Année académique 2017-2018

Nous tenons à remercier nos deux promoteurs, Sandra Soarez-Frazão et Etienne Rivière sans qui ce mémoire n'aurait pas été possible. Leurs conseils avisés, leurs suivis tout au long de l'année et leurs connaissances dans leur domaine respectif, le génie civil et l'informatique, nous ont permis de mener à bien ce mémoire.

Nous voulons également remercier Damien François et Jonathan Lambrechts qui ont acceptés d'être nos lecteurs. Ils sont tous deux spécialisés dans une facette différente de notre mémoire, respectivement celle du calcul intensif et celle de la mécanique des fluides.

Nous souhaitons ensuite remercier le CÉCI pour leurs formations sur le calcul parallèle et plus particulièrement à l'usage des clusters. Nous les remercions également pour la mise à disposition des clusters pour la validation nos tests.

Nous tenons à remercier Nicolas Detienne de l'ICTM pour l'accès à l'ordinateur qui nous a permis de tester nos programmes multithreadés.

Et pour finir, nous remercions Guillaume Derval pour son aide et ses réflexions sur le partitionnement du maillage en régions et plus particulièrement pour l'usage de Gurobi.

Le programme SFlow2D, développé par le Pr S. Soares Frazão lors de sa thèse, est un outil de modélisation et de prévision de crues pour des topologies complexes. La simulation à grande échelle engendre un coût calculatoire important. Le but de ce mémoire est d'augmenter la vitesse de calcul par la parallélisation sans modifier la physique du calcul. Les trois méthodes de parallélisation étudiées sont le multithreading, l'usage d'un cluster et l'emploi de carte graphique. Les enjeux qui garantissent la diminution des temps de calcul sont la répartition efficace des calculs, le partage d'informations et la centralisation des résultats. De plus, la répartition du travail au sein d'un cluster nécessite une étape supplémentaire : le partage du domaine en sous-régions. La division doit être équilibrée tout en minimisant la taille des frontières entre les régions réduisant le nombre d'informations échangées. Il s'agit d'un problème NP-difficile. Deux cas de tests réels valident nos résultats et quantifient l'amélioration du temps de calcul. Le maillage du premier est composé de 11 000 cellules et représente une simulation d'une minute. Le second étudie une rupture de barrage. Il possède plus de 60 000 cellules et simule une crue de plus de 40 heures. Initialement, le programme SFlow2D s'exécute, respectivement, en 2 minutes et en 54 heures. La carte graphique ne produit pas de résultats concluants tandis que l'emploi d'un cluster du CÉCI sur la rupture de barrage a permis de réduire le temps à moins d'une heure avec 48 unités de calculs. Un ordinateur fournit, grâce au multithread, le résultat en moins de 15 heures (avec 8 threads).

The SFlow2D program, developed by Pr. S. Soares Frazão, is a hydraulic model to simulate shallow water flows on complex topologies. Hydraulic models used on large-scale, and high resolution problems are computationally expensive. The purpose of this work is therefore to investigate three parallelization methods to reduce execution. These methods are multithreading, the cluster computing and GPU programming. The main challenges are efficient work distribution, information synchronization and centralizing results. Furthermore, work distribution on a cluster requires an additional step : the partitioning of the domain into smaller regions. A good balance in size between regions and minimized borders between regions of the partition allow for reduced data exchange. This partitioning problem has been proven to be NP-hard and has been solved using a heuristic. Two test cases are used in order to validate and quantify the speed-up of our developments. The first is composed of 11,000 cells and represents a simulation of one minute which takes less than 2 minutes to complete. The second case study is based upon the failure of a dam and is composed of more than 60,000 cells. It simulates a flood event of more than 40 hours and takes 54 hours to be generated. Using the cluster has allowed us to reduce the execution time to less than an hour with 48 computational cores. The multithreaded version, takes less than 15 hours to complete (using 8 threads), while the GPU version doesn't produce concluding results on the Tous test case.

Table des matières

Introduction **1**

1 Fondements physiques **2**
1.1 Simulation par le cas de test . . . 2
1.2 Discrétisation numérique de l'écoulement . . . 3
1.2.1 Méthodes des volumes finis . . . 4
1.3 Étapes d'exécution du calcul . . . 4
1.4 État de l'art . . . 5
1.4.1 Technologies spécifiques à la simulation d'écoulement . . . 5
1.4.2 Gestion de la mémoire . . . 6
1.4.3 Démarches de parallélisation . . . 6
1.4.4 Résultats . . . 7

2 Analyse du Logiciel **9**
2.1 Analyse générale du Code . . . 9
2.1.1 Structure et subdivision du code . . . 9
2.1.2 Étapes d'exécution du programme . . . 10
2.1.3 Benchmarking de l'exécution des fonctions . . . 11
2.1.4 Parties du programme parallélisables . . . 13
2.1.5 Limite théorique d'accélération . . . 13
2.1.6 Calcul du gain et de l'efficacité . . . 14
2.2 Validation des résultats . . . 14
2.2.1 Un problème de précision . . . 14

3 Division en régions **17**
3.1 Notations et propriétés . . . 18
3.2 Le problème de partitionnement . . . 18
3.2.1 Le partitionnement du domaine de calcul . . . 18
3.2.2 Un problème équivalent . . . 19
3.2.3 Un problème NP-difficile . . . 20
3.3 Gurobi Optimizer . . . 20
3.3.1 Résultats . . . 21
3.4 Méthode approchée . . . 21
3.4.1 Matrice des distances . . . 21
3.4.2 Centroïdes . . . 23
3.4.3 Attribution initiale des cellules . . . 24
3.4.4 Amélioration de l'attribution des cellules . . . 24
3.4.5 Recherche du meilleur partitionnement . . . 28
3.5 Résultats . . . 29
3.5.1 Amélioration d'une solution initiale . . . 29
3.5.2 Comparaison : distance euclidienne ou matrice des distances . . . 29
3.5.3 Test sur un grand domaine . . . 32
3.6 Tests unitaires . . . 33
3.6.1 Chemin entre deux cellules . . . 34
3.6.2 Déplacement des cellules entre régions . . . 34
3.7 Conclusion . . . 35

4 Multithreading **36**
4.1 Concepts de parallélisation . . . 36
4.2 Outils de parallélisation . . . 39
4.2.1 OpenMP . . . 39
4.2.2 Pthread . . . 40
4.3 Optimisations du code . . . 41
4.3.1 Vectorisation . . . 41
4.4 Parallélisation . . . 42
4.4.1 Flux . . . 42
4.4.2 Update . . . 43
4.4.3 dtMin . . . 43
4.4.4 Écriture de fichiers . . . 44
4.4.5 Thread pool . . . 44
4.5 Résultats et Analyse . . . 45
4.5.1 Benchmark . . . 45
4.5.2 Parallélisations supplémentaires . . . 46
4.6 Notes sur l'état de l'art . . . 47

5 Parallélisation sur cluster **48**
5.1 Théorie . . . 48
5.1.1 Les messages entre deux nœuds . . . 48
5.1.2 Messages collectifs . . . 49
5.2 Adaptation de SFlow2D . . . 50
5.2.1 Division du domaine . . . 50
5.2.2 Initialisation . . . 52
5.2.3 Itérations . . . 53
5.2.4 Fichiers de sortie . . . 55
5.3 Résultats . . . 55
5.3.1 Toce . . . 56
5.3.2 Tous . . . 57

6 Programmation parallèle sur Processeur Graphique **60**
6.1 Architecture générale du GPU . . . 60
6.2 Choix du framework de développement sur GPU . . . 61
6.2.1 Les Frameworks . . . 61
6.2.2 Langage de programmation . . . 61
6.3 Implémentation sur GPU . . . 62
6.3.1 Fonctionnement de CUDA . . . 62
6.3.2 Modèle de Programmation CUDA . . . 62
6.4 Résultats . . . 66
6.4.1 Limitations . . . 66
6.5 Améliorations futures . . . 67

Conclusion **68**

Bibliographie

Annexe **I**
Abstract conférence mondiale IAHR . . . I

Liste des abréviations

CÉCI	Consortium des Équipements de Calcul Intensif
CPU	Central Processing Unit (Processeur)
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit (Processeur graphique)
GPGPU	General Purpose Graphics Processing Unit (Calcul accéléré sur processeur graphique)
IEEE	Institute of Electrical and Electronics Engineers (Institut des ingénieurs électriciens et électroniciens)
ICTM	Institute of Information and Communication Technologies, Electronics Mathematics and Applied (Institut de l'information et des technologies de communications, l'Électronique et Mathématiques appliquées)
LP	Linear programming (Programmation linéaire)
MIP	Mixed-Integer Programming (Programmation d'entiers mixte)
MPI	Message Passing Interface (Interface de transmission de message)
NP	Non-deterministic Polynomial time (Temps polynomial non déterministe)
POO	Programmation orientée objet
POSIX	Portable Operating System Interface (Interface de système d'exploitation portable)
RAM	Random Access Memory (Mémoire dynamique)
SIMD	Single Instruction Multiple Data (Instruction unique données multiples)
SIMT	Single Instruction Multiple Threads (Instruction unique threads multiples)
SPMD	Single Program Multiple Data (Programme unique données multiples)
UMA	Unified Memory Access (Accès mémoire unifié)

Table des figures

1.1 Instantané de la simulation du cas test Toce au temps t = 28 s 2
1.2 Schéma itératif de mise à jour de la hauteur d'eau 4
1.3 Structure de l'exécution et schéma du domaine 5

2.1 Représentation simplifiée du domaine de calcul et des différents outils de mesure 10
2.2 Étapes d'exécution du programme 11
2.3 Différence de hauteur entre instantanés de la simulation du cas test Toce 15
2.4 Cellules affichant des différences de hauteur importantes 16

3.1 Étapes d'exécution de la méthode approchée 22
3.2 Domaine avec trois régions 25
3.3 Schéma général du partitionnement 29
3.4 Résultat de l'amélioration du partitionnement initial 29
3.5 Partition de Toce à l'aide de la distance euclidienne 31
3.6 Partition de Toce à l'aide de la matrice des distances 32
3.7 Partitionnement du maillage de Tous en 24 régions 33
3.8 Domaine pour les tests des chemins 35

4.1 Décomposition en domaine 36
4.2 Différence entre partage en bloc et partage cyclique 37
4.3 Décomposition fonctionnelle 37
4.4 Mauvaise distribution des tâches 38
4.5 Le cycle des threads 42

5.1 Étapes du calcul et de l'envoi d'échange de données par MPI 51
5.2 Domaine d'exemple 51
5.3 Échanges d'information à l'interface de 2 régions 53
5.4 Schéma itératif d'un nœud du cluster 54
5.5 Format du type du message transportant les informations pour la génération des fichiers de sortie 55
5.6 Graphe du temps de calcul par rapport au nombre de nœuds pour Toce 56
5.7 Graphe du temps de calcul par rapport au nombre de nœuds pour Tous 58

6.1 Architecture simplifiée du CPU et du GPU NVIDIA 60
6.2 Grille de blocs de threads 62
6.3 Schéma simplifié des informations d'une cellule dans SFlow2D 65
6.4 Différence entre structure de tableaux et tableau de structures 65
6.5 Schéma itératif sur deux streams 65

Liste des tableaux

1.1 Statistiques du barrage Tous 3
1.2 Synthèse des indications, limitations et résultats de la parallélisation dans la littérature scientifique 8

2.1 Temps d'exécution des fonctions principales appelées par le programme SFlow2D 12
2.2 Mesure du temps d'exécution : programme de base 13

3.1 Taille des régions générées à l'aide de la distance euclidienne pour Toce 30
3.2 Taille des régions générées à l'aide de la matrice des distances pour Toce 30
3.3 Evolution de la taille des frontières avec la distance euclienne pour Toce 31
3.4 Évolution de la taille des frontières à l'aide de la matrice des distances pour Toce 31
3.5 Taille des régions du partitionnement de Tous 34
3.6 Évolution de la taille des frontières pour Tous 34

4.1 Temps d'exécution décomposé des différentes versions améliorées et l'accélération par rapport au programme initial -O0 46
4.2 Temps d'exécution des versions les plus performantes de OpenMP avec les parallélisations supplémentaires sur Toce 47
4.3 Temps d'exécution des versions les plus performantes avec les parallélisations supplémentaires sur Tous 47

5.1 Temps d'exécution de Toce avec la distance euclidienne avec -O0 57
5.2 Temps d'exécution de Tous avec la distance euclidienne avec -O3 58

6.1 Propriétés principales d'OpenACC, OpenMP, OpenCL, et CUDA 61
6.2 Avantages et inconvénients de l'accès unifié en mémoire (UMA) 64
6.3 Caractéristiques de la carte graphique GeForce GTX 770 66
6.4 Temps d'exécution moyen en fonction de la configuration Threads/Nombre de blocks 66

Introduction

Le programme SFlow2D a été développé par le Pr S. Soares Frazão dans le cadre d'une thèse réalisée au sein du département d'hydraulique de l'unité de Génie Civil et Environnemental de l'Université Catholique de Louvain-la-Neuve. Ce programme, implémenté en C++ à l'aide des libraires standards, modélise l'écoulement de l'eau pour des topographies complexes.

La simulation d'écoulement est primordiale dans les zones habitées qui présentent un risque d'inondations dues à des pluies abondantes ou des ruptures de barrage. La comparaison de ces simulations avec des cas de tests aux mesures connues est importante : ils permettent de valider et donner du poids aux modèles mathématiques. Ces modèles pourront à leur tour identifier préventivement de nouvelles zones à risques et des mesures à prendre en cas de rupture de barrage ou de pluie abondante. Effectuer des simulations réelles permet d'obtenir des données fiables.

Le temps d'exécution d'une simulation va dépendre du temps simulé et du raffinement de la reconstruction de la topologie appelée maillage. Initialement, il existe une correspondance linéaire entre le temps d'exécution et celui de l'évènement. Une topologie complexe additionnée à un temps simulé important impliquent des temps d'exécution de quelques heures à plusieurs jours, qui sont plus longs que la réalité.

L'objectif de ce mémoire est donc de réduire le temps d'exécution du programme SFlow2D sans modifier le cœur du calcul, c'est-à-dire la physique du problème. La réduction du temps d'exécution se fait à l'aide de la parallélisation des calculs. La parallélisation consiste à effectuer différentes opérations en même temps réparties sur les différents cœurs de calcul de telle sorte que tous les cœurs effectuent la même charge de travail. Trois pistes ont été explorées : le calcul parallèle sur *CPU* (processeur ou unité centrale de traitement), sur *cluster* (ensemble d'ordinateurs reliés entre eux, appelés nœuds) et sur un *GPU* (processeur graphique) [1].

Afin de répartir uniformément le travail effectué par chaque nœud d'un cluster, une étape supplémentaire de précalcul a été ajoutée à l'exécution du programme. L'étape de précalcul subdivise le maillage en sous-régions de tailles équivalentes dont les frontières avec d'autres sous-régions sont les plus petites possible.

Dans un soucis de pouvoir modifier dans le futur la physique du problème et y intégrer aisément des nouvelles améliorations, nous donnons de nombreux détails dans les parties les moins évidentes de nos implémentations. Nous présentons également pour chaque technologie les différents concepts que nous avons employés.

En résumé, le mémoire se compose tout d'abord d'un premier chapitre introduisant les concepts théoriques nécessaires à la compréhension et la situation du problème ainsi que l'état des travaux apparentés à la thématique. Le chapitre 2 présente une analyse du logiciel tel qu'il a été fourni. Le chapitre 3 traite de la division du domaine en petites régions. Les différentes techniques de parallélisation sont abordées dans les trois chapitres suivants. Ceux-ci expliquent tout d'abord les technologies en question et ensuite les développements et améliorations liées à ces nouveaux concepts. Enfin, un dernier point est consacré aux résultats obtenus grâce à l'intégration de ces technologies. Le dernier chapitre conclut ce travail et apporte quelques perspectives pour de nouvelles recherches.

1

Fondements physiques

Les simulations numériques d'inondations résultant d'évènements catastrophiques comme des ruptures de barrage font l'objet de nombreuses recherches afin d'améliorer l'efficacité des codes de calcul utilisés à cet effet. Notamment, il est nécessaire de pouvoir disposer de résultats en un temps raisonnable lorsqu'il s'agit d'évaluer les risques pour les populations voisines. Le présent mémoire s'inscrit dans cet effort en explorant diverses pistes d'accélération d'un code de calcul existant (SFlow2D) via la parallélisation de celui-ci.

Le programme SFlow2D, implémenté en C++ à l'aide des libraires standards, modélise l'écoulement de l'eau pour des topographies complexes sur base de la méthode des volumes finis en deux dimensions. La topographie est reproduite à l'aide d'un maillage non structuré dont un élément individuel, un triangle sur la figure 1.1, est appelé une cellule.

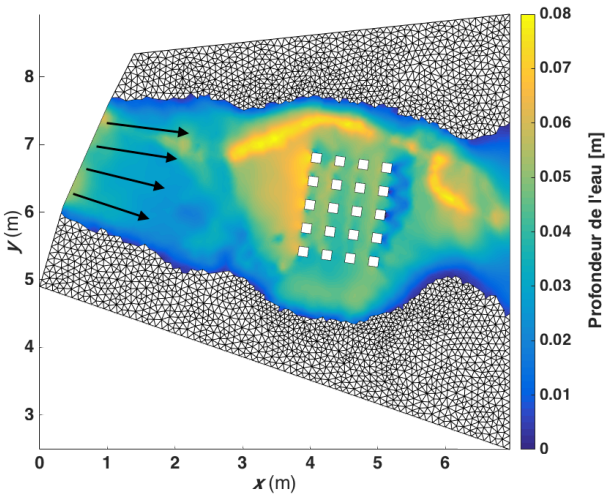


FIGURE 1.1 – Instantané de la simulation du cas test Toce au temps t = 28 s

Ce chapitre permet d'introduire les notions et le vocabulaire liés à la simulation d'écoulement par le programme SFlow2D. Le premier point présente les cas de tests utilisés pour valider les développements. Ensuite, la seconde section explique le schéma numérique de la simulation. Le troisième point est consacré aux étapes d'exécution du calcul. Finalement, le dernier point fait l'état des travaux apparentés à la parallélisation d'outils de simulation hydraulique.

1.1 Simulation par le cas de test

La figure 1.1 est un instantané de la simulation de l'écoulement d'un cas de laboratoire à petite échelle, le cas test *Toce* présenté dans l'article [2]. Il s'agit d'un modèle réduit représentant une vallée située en Italie, qui a fait l'objet d'études dans le cadre d'un projet européen. L'écoulement simulé correspond à une rupture de barrage, ce qui signifie que les cellules du maillage sont

initialement « sèches » et elles vont progressivement être envahies par l'eau au fil de la simulation.

Sur la figure 1.1, la profondeur de l'écoulement est représentée en couleur allant du bleu, correspondant à une profondeur faible, au jaune, une profondeur plus importante. Le maillage du reste du domaine de calcul représente l'arrière-plan de l'image, il s'agit de cellules sèches à cet instant-là. Dans le cas test Toce, le domaine est relativement petit (~11 000 cellules) et la simulation dure une minute correspondant au temps simulé (à ne pas confondre avec le temps nécessaire pour générer la simulation qui est le temps d'exécution ou de calcul). Ce cas test sera donc utilisé pour tester les différents développements et améliorations apportées au code.

Afin de valider les améliorations et de quantifier l'accélération du calcul, un cas test plus complexe et à plus grande échelle est utilisé. Il s'agit du cas de la rupture du barrage du Tous en Espagne [3]. Ce barrage s'est rompu en 1982 suite à de fortes pluies (plus de 600 x 10^6 m^3) ce qui a provoqué un excès de la capacité du barrage de 120 millions de m^3. La vague résultant de cette rupture a eu des effets catastrophiques : 300 km^2 de terres habitées, y compris de nombreuses villes et de nombreux villages, ont été inondés affectant pas moins de 200 000 personnes dont 100 000 ont dû être évacuées, pour un total de huit victimes. La petite ville de Sumacarcel a été complètement submergée avec des profondeurs d'eau dans les rues atteignant jusqu'à 4 m. Pour ce cas, le maillage comporte environ 61 000 cellules et la crue à simuler dure 40 heures. La table 1.1 reprend quelques statistiques supplémentaires du barrage Tous.

Hauteur en crue	98,5m (MSL)
Élévation de l'eau en opération normale	84 m
Hauteur des évacuateurs de crues (portes ouvertes)	87,5 m
Hauteur des évacuateurs de crues (portes fermées)	87,5 m
Surface du bassin versant du barrage	17 820 km^2
Surface du bassin de la rivière Júcar	21 600 km^2
Capacité de réservoir en opération normale	52 x 10^6 m^3
Capacité de réservoir en crue	122 x 10^6 m^3

TABLE 1.1 – Statistiques du barrage Tous

1.2 Discrétisation numérique de l'écoulement

La discrétisation d'une topologie complexe par sa représentation sous forme de maille permet d'y définir des informations spécifiques à chaque cellule. Dans le cas de Tous et de Toce, la simulation s'effectue sur un maillage composé de triangles. Le programme est également conçu pour fonctionner avec d'autres formes géométriques qui possèdent plus de côtés. Chacune des cellules est composée de plusieurs sommets, les *nœuds*, et de côtés, les *interfaces*. Une cellule est caractérisée par plusieurs données dont les plus importantes sont les variables hydrauliques : la hauteur de l'eau et le débit de l'eau.

Le domaine de calcul comprendra donc l'entièreté des cellules, leurs nœuds et leurs interfaces. De plus, il est possible d'y définir les outils de mesure suivants :

— Les *capteurs* mesurent les données hydrauliques des cellules sur lesquels ils ont été placés.

— Les *sections de mesure de débit* regroupées.

— Un appareil photo qui produit des *instantanés* de l'écoulement à des repères dans le temps prédéfinis.

La figure 1.3b schématise le domaine de calcul et permet la compréhension de l'intuition physique. Le chapitre suivant aborde l'implémentation du programme SFlow2D et présente une

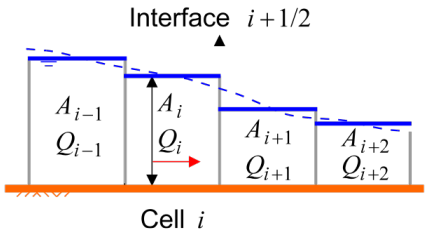
version plus complète du domaine de calcul à la figure 2.1.

1.2.1 Méthodes des volumes finis

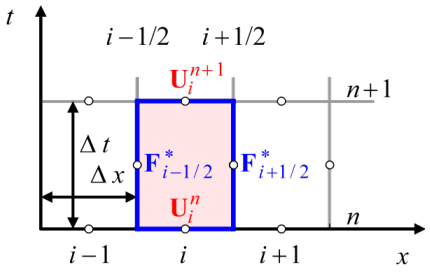
La simulation elle-même est calculée par la méthode des volumes finis. Cette méthode emploie un modèle explicite, c'est-à-dire que l'itération suivante n'a que besoin des informations de l'itération la précédant. Elle se définit à l'aide de l'équation suivante :

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1/2} - F_{i-1/2}\right) + S_i \Delta t \tag{1.1}$$

où $F_{i+1/2}$ et $F_{i-1/2}$ sont les flux entrant et sortant aux interfaces *i+1/2* et *i-1/2* de la cellule i à l'itération n, Δx est le rayon du plus petit cercle inscrit dans une des cellules du maillage, Δt est le pas de temps. La figure 1.2a présente un exemple type de l'évolution du débit de l'eau. La figure 1.2b présente un schéma type de l'évolution de la hauteur de l'eau U de la cellule i à l'itération n+1 à partir de l'itération n.



(a) Aire A et débit Q des cellules i-1 à i+2



(b) Mise à jour de la hauteur de l'eau à l'itération n+1 de la cellule i à partir l'itération n

FIGURE 1.2 – Schéma itératif de mise à jour de la hauteur d'eau

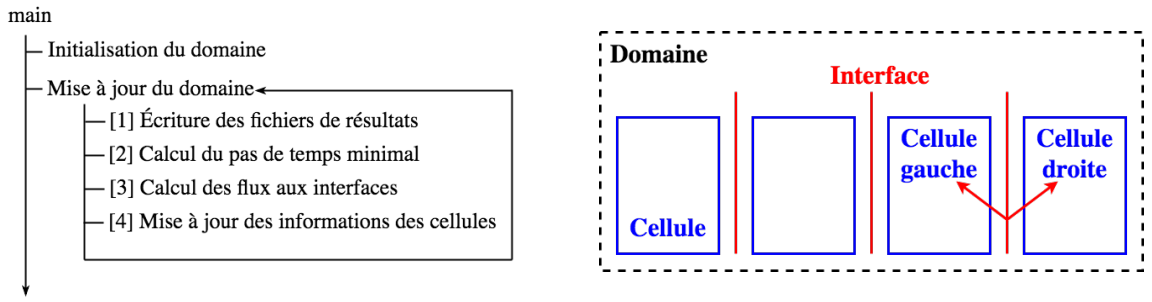
1.3 Étapes d'exécution du calcul

L'exécution de la simulation est répartie en deux grandes étapes : l'initialisation du domaine de calcul et sa mise à jour (voir section 2.1.2). Une fois que l'initialisation du domaine est terminée, le programme effectue la simulation de l'écoulement. Les mises à jour du domaine se font dans un intervalle prédéfini par l'utilisateur [$T_{début}$, T_{fin}] correspondant à la durée de la simulation.

Une itération est composée de plusieurs étapes :

1. Le temps actuel de la simulation est comparé au prochain repère dans le temps marquant l'écriture des fichiers de résultats. S'ils sont égaux, alors le programme enregistre les variables hydrauliques en des points précis, effectue un instantané de la simulation au temps actuel et mesure le débit à travers les sections surveillées.
2. Le calcul du pas de temps minimal pour l'itération actuelle. Pour ce faire, il faut parcourir l'ensemble des cellules et calculer leur pas de temps en employant leurs variables hydrauliques et leurs dimensions géométriques. Il faudra sélectionner le plus petit pas de temps parmi toutes les cellules. Si aucune des cellules ne possède un niveau d'eau minimal, le pas de temps de l'itération prend la valeur d'un paramètre fixé.
3. Évaluation des flux échangés aux interfaces entre les cellules gauches et droites, mais également aux interfaces qui forment la frontière du maillage.
4. Les variables hydrauliques des cellules sont ensuite mises à jour.

Le nombre d'itérations n'est pas prédéfini en avance. De petits pas de temps engendrent un grand nombre d'itérations par rapport à des pas de temps longs. De plus, une quantité de cellules importante, liée à de très longs évènements, engendrent des temps de simulation très longs.



(a) Structure de l'exécution du calcul

(b) Schéma du domaine de calcul

FIGURE 1.3 – Structure de l'exécution et schéma du domaine

Paralléliser les calculs et plus particulièrement dans le cas de maillages non structurés nécessite de porter attention aux informations partagées lors des calculs. Dès lors, avant d'appliquer une quelconque modification du code, il est important d'effectuer l'analyse du fonctionnement et de l'exécution de SFlow2D. Cette étape offre donc une meilleure compréhension du programme et ouvre la voie aux différentes améliorations par la parallélisation qui réduiront son temps d'exécution.

L'analyse de la structure du domaine et de l'interaction entre ses différents éléments (détaillée dans le chapitre 2) a mis en évidence qu'une étape de précalcul doit être ajoutée à l'initialisation du domaine dans le cas de la parallélisation sur un cluster. Cette étape supplémentaire est également connue sous le nom du problème de partitionnement. Elle consiste à subdiviser le domaine en plus petites régions de taille égales minimisant la taille des frontières entre les nouveaux sous-domaines formés. Réduire la taille des frontières communes diminue la quantité d'informations à synchroniser. De plus, il est primordial que la synchronisation des données se fasse de façon à éviter les problèmes de mise à jour lorsqu'un ou plusieurs cœurs de calculs tentent de mettre à jour les mêmes données par exemple au sein d'une même cellule.

1.4 État de l'art

Le problème de la simulation efficace d'un écoulement par la résolution des équations d'eaux peu profondes à l'aide de la méthode des volumes finis a été étudié à de nombreuses reprises. La parallélisation en employant un GPU ou un cluster sont les deux méthodes offrant les résultats les plus concluants. Toutefois, ils requièrent le plus d'adaptation au niveau de l'implémentation, une analyse détaillée de la structure des données et de l'exécution par rapport à un code séquentiel.

Cette section synthétise les informations venant de la littérature : les indications concernant l'usage de la mémoire, les technologies se prêtant au mieux à la parallélisation du calcul, et l'intégration de ces technologies. Le tableau 1.2 récapitule les méthodes de parallélisation.

1.4.1 Technologies spécifiques à la simulation d'écoulement

La parallélisation par l'utilisation de plusieurs cœurs ou par le GPU s'effectue en étendant une implémentation séquentielle à l'aide de librairies spécifiques. Dans le multithread, il est possible d'utiliser la libraire *pthread* ou bien d'ajouter des directives pour indiquer au compilateur les

endroits parallélisables par exemple avec OpenMP. Zhang et al. [4] concluent que l'utilisation d'OpenMP se prête bien à la simulation d'écoulement. OpenMP est également un compromis entre facilité d'implémentation et efficacité au niveau de l'accélération. OpenACC, tout comme OpenMP, spécifie un ensemble de directives au compilateur et des variables d'environnement pour faciliter l'introduction de parallélisation dans un programme. Cependant, OpenACC est utilisé spécifiquement dans la parallélisation sur des systèmes hétérogènes, c'est-à-dire des systèmes combinant l'utilisation du CPU et du GPU.

Il existe plusieurs pistes pour l'implémentation de SFlow2D sur GPU. La première est l'ajout de directives OpenMP ou OpenACC au code pour confier une partie du travail au GPU. La seconde piste est l'utilisation d'un langage de programmation adapté tel qu'OpenCL ou CUDA. Dans l'article de Lacasta et al. [5] et dans le benchmark effectué par Memeti et al. [6], la plateforme de programmation parallèle CUDA est la solution la plus efficace. Cependant, elle nécessite l'utilisation d'une carte graphique de la marque NVIDIA, la seule capable d'exécuter du code CUDA.

1.4.2 Gestion de la mémoire

La gestion du placement des données en mémoire est la clé d'un calcul accéléré performant.

1. Pour le CPU, le concept d'une seule opération sur plusieurs données (SIMD) s'implémente par la transformation de données en vecteur. Les calculs se font dès lors par des opérations vectorielles. Neal et al. [7] concluent que l'utilisation de la vectorisation n'est pas efficace lorsqu'il y a une combinaison de cellules sèches et de cellules sous eau, car les calculs sont différents.
2. Les structures de données efficaces sur GPU doivent permettre des accès parallèles rapides ainsi que des itérations parallèles tout en respectant les contraintes de l'architecture mémoire de celui-ci. Lacasta et al. [5] et Castro et al. [8] privilégient une représentation sous forme de tableaux de structure. Le but est de diminuer la quantité d'informations utilisées lors de chaque itération.
3. Dans un cluster, Lai et Khan [9] et Sanders et al. [10] suggèrent de partager le domaine de calcul en régions de taille identique et dont la frontière générale est la plus petite possible. Cette division peut se faire à l'aide la librairie *Metis*. Dans ces deux articles, la distribution des informations se fait au moyen de fichiers spécifiques contenant les données initiales. Chaque nœud du cluster possédera trois fichiers pour, respectivement, les sommets, les interfaces et les cellules ainsi que différents fichiers pour permettre la synchronisation des données entre les régions. Il est important de noter qu'une région doit connaître les cellules qui lui sont adjacentes [7].

1.4.3 Démarches de parallélisation

Le processus de parallélisation d'un programme est décrit dans l'article de Martinez et al. [11] pour le GPU et dans l'article de Zhang et al. [4, 12] pour OpenMP et MPI (libraire de communication employée dans un cluster). Il est recommandé d'effectuer une analyse approfondie du programme afin d'identifier les endroits parallélisables.

Pour le GPU, le programmeur détermine d'abord les portions du programme parallèle et segmente le programme sous forme de sections indépendantes parallèles dont les entrées et résultats sont des tableaux de données. Ensuite, il adapte le programme pour incorporer les étapes clés du calcul sur GPU décrites dans le tableau 1.2. L'article *A Survey of General-Purpose Computation on Graphics Hardware* [13] reprend une explication plus détaillée des étapes clés et sert de guide à la parallélisation sur GPU.

Dans une parallélisation utilisant MPI, il est important de réduire la quantité d'informations échangées. Par conséquent, l'article de Neal et al. [7] suggère d'utiliser la fonction `Allreduce` pour le calcul du pas de temps minimal. Chaque région calcule son pas de temps minimal et l'envoie. `AllReduce` applique une opération, dans ce cas-ci le calcul du minimum, sur toutes les informations reçues et renvoie le résultat à toutes les régions.
La communication et la synchronisation des données communes à deux régions doivent se faire efficacement et correctement. L'article de Neal et al. [7] propose une démarche pour la mise à jour du domaine formé d'un maillage régulier décomposé en sous-régions réparties sur plusieurs nœuds du cluster. Le modèle de parallélisation décrit une procédure pour la synchronisation des données sur tous les nœuds. La synchronisation se résume aux étapes suivantes :

1. Chaque région calcule le flux échangé pour chaque interface.
2. La région envoie le flux de toutes les interfaces adjacentes qui forment la frontière de la région.
3. Attente de l'envoi et de la réception de tous les flux échangés entre les régions.
4. Mise à jour des niveaux d'eau et du débit dans chaque cellule.
5. Envoi des valeurs des niveaux d'eau des cellules adjacentes aux régions.

1.4.4 Résultats

Les trois méthodes de parallélisation accélèrent le calcul en fonction du nombre de cœurs de calculs disponibles. Le multithread est limité à la quantité de cœurs disponibles dans un ou plusieurs processeurs, MPI est limité par la communication entre les régions dans le cas de SFlow2D. Zhang et al. [4] établissent que si le nombre de cellules est suffisamment élevé, alors l'accélération évolue linéairement en fonction du nombre de cœurs de calculs. Le GPU offre potentiellement des gains d'accélérations très importants, mais nécessite un développement laborieux, des structures de données optimisées et une analyse détaillée des opérations vectorisables.

Le tableau 1.2 synthétise les points importants tirés des différents articles qui ont permis de garantir aux auteurs une parallélisation efficace. Les différents conseils se répartissent en trois catégories : les contraintes et limitations liées au type de parallélisation, le potentiel du support de parallélisation et les langages de programmation qui se prêtent au mieux à la parallélisation.

	Multithread	GPU	Cluster
Mémoire	Porter attention au type de mémoire (partagée ou privée) pour les différents threads [12]	Tableau de structures de données [5] Optimiser pour éviter les accès concurrents des threads et minimiser la synchronisation entre les exécutions parallèles [11] Minimisation des accès à la mémoire globale [13, 11] Réordonner les cellules et les interfaces [5]	Un programme unique, données multiples [9] Partitionnement du domaine [9, 10]
Points importants	Répartition du travail entre les threads [4] Identification des endroits intéressants à paralléliser [4]	Étapes clés : allocation de mémoire GPU et transfert des données → calcul → copie en mémoire [11]	Fonction optimisée spécifiques à MPI (ex : calcul du pas de temps minimal avec `AllReduce`) [7] Communication entre les régions [9, 7]
Technologie	OpenMP est un bon choix de technologie (bon rapport entre simplicité et efficacité) [4] SIMD non-efficace si les cellules sèches sont mélangées aux cellules sous eau [7]	Utilisation framework propriétaire CUDA plus efficace comparé aux autres technologies (OpenCL, OpenAcc, OpenMP)	
Résultats	Accélération de x6,37 pour 8 threads et 8,15 pour 16 threads dans un grand domaine composé de quadrilatères [4] OpenMP fournit des résultats similaires à MPI jusqu'à 4 cœurs et est plus facile à développer [7]	Accélération importante x20 à x120	Accélération linéaire à l'augmentation de cellules s'il y a suffisamment de cellules dans le domaine [14]
	Accélération variable en fonction du type de maillage (structuré ou non, triangles ou carrés, quantité de cellules)		

Table 1.2 – Synthèse des indications, limitations et résultats de la parallélisation dans la littérature scientifique

2
Analyse du Logiciel

Pour paralléliser efficacement un logiciel, il faut tout d'abord l'étudier en profondeur. Ce processus se fait en plusieurs étapes. Dans un premier temps, il faut décomposer les appels aux méthodes du programme. La décomposition analysera le temps d'exécution moyen d'un appel, le nombre d'appels à ces méthodes et toutes autres méthodes imbriquées. À partir de cette décomposition, il sera possible de mettre en évidence les parties du programme où plusieurs *threads* (fils d'exécutions légers) seraient susceptibles de modifier les mêmes ressources au même moment. Dans un second temps, les résultats de la décomposition seront utilisés afin d'approximer l'accélération maximale possible à l'aide de la loi d'Amdahl [15]. Finalement, nous abordons les problèmes de précisions et proposons une démarche de validation des résultats.

2.1 Analyse générale du Code

Le programme SFlow2D a été conçu en plusieurs classes suivant le paradigme de programmation orientée objet (POO). Le principe de base de la POO repose sur la division du problème en un ensemble de petits blocs, appelés objets. Chaque objet, défini par sa classe, est un conteneur d'informations qui lui sont propres. Ses informations portent le nom d'attributs. Un objet d'une classe se dit instance de cette classe. Chaque classe implémente des fonctions qui s'appliquent sur l'objet et modifient les attributs de cet objet. La POO permet ainsi de manipuler aisément les équations mathématiques en code C++.

2.1.1 Structure et subdivision du code

Le principe de décomposition de problème en petits sous-problèmes est très utile dans le cas d'une simulation d'écoulement hydraulique où de nombreuses informations sont échangées au fur et à mesure des itérations (comme le montre la figure 1.1).

Une instance de classe `CDomain` correspond à l'entièreté de la représentation informatique du domaine de calcul. Le maillage reproduisant la topologie complexe est défini par ses trois composantes : les cellules, les nœuds et les interfaces. Chacune de ces trois composantes est implémentée par sa classe :

— Une cellule est représentée par une instance de la classe `CCell`. Chacune d'elle possède des propriétés et des données hydrauliques telles que la hauteur de l'eau et le débit. Ces informations sont les attributs de l'objet cellule.

— Une interface est représentée par une instance de la classe `CInterface`. Il en existe plusieurs types implémentés sous la forme d'une sous-classe. Parmi tous les types d'interfaces possibles, ceux utilisés dans Toce sont `Ci_RoeS0upwind` , `CiWall`, `CiHydrogram` et `CiTransmissive`.

— Un nœud est représenté par une instance de la classe `CNode`.

La classe `CDomain` définit les outils de mesures suivants :

— L'ensemble de ces capteurs est regroupé par une instance de la classe `CGauge`.

— Les *mesures de débit sur une section* sont regroupées dans une instance de la classe `CMonitorSection`.

— Les *instantanés* sont enregistrés à l'aide de la classe `CPicture`.

La figure 2.1 schématise les différents objets qui se combinent pour modéliser le domaine de calcul au sein d'une instance de la classe `CDomain`.

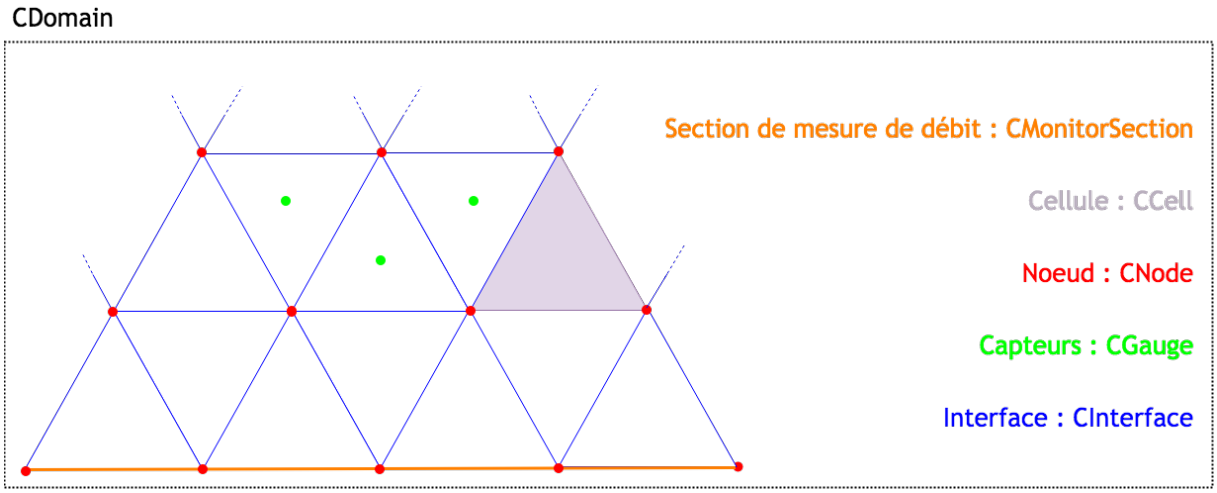


FIGURE 2.1 – Représentation simplifiée du domaine de calcul et des différents outils de mesure

Avant d'analyser l'exécution du programme, notons que la structure générale est détaillée dans le fichier `Classdef.h`. Ce fichier contient les variables, méthodes et classes utilisées dans le programme. Finalement, il existe encore le fichier `COutput.cpp` où les fonctions d'écriture de fichiers sont définies, et le fichier `main.cpp` pour exécuter le programme.

2.1.2 Étapes d'exécution du programme

L'exécution du programme, comme présenté sur la figure 2.2 et plus en détail dans la table 2.1, est composée d'une étape d'initialisation et d'une étape de mise à jour du domaine.

Dans la phase d'initialisation, la classe CDomain lit le fichier contenant les paramètres de la simulation. Ce fichier, rempli par l'utilisateur, définit le temps de simulation, les fichiers reprenant les nœuds, cellules et interfaces, les fichiers détaillant la configuration des outils de mesure ainsi que d'autres paramètres utiles au bon fonctionnement des équations physiques.

La mise à jour du domaine se calcule par itération de la boucle principale. Les itérations se succèdent jusqu'à couvrir l'entièreté du temps de simulation. Une itération est organisée en plusieurs sous-étapes traduites par les appels de fonctions suivants :

1. Comparaison du temps actuel au prochain de repère temporel d'écriture de résultats : {`classe de l'outil de mesure`}→`CheckTime()`. S'ils sont égaux alors le programme enregistre les variables hydrauliques en des points précis, effectue des enregistrements de l'état des variables pour une visualisation ultérieure des résultats et mesure le débit à travers les sections surveillées.
2. Le pas de temps minimal est calculé par `dtMin()`.
3. Les flux sortants et rentrants sont calculés à chacune des interfaces par l'appel à `flux()`
4. La mise à jour des variables hydrauliques de toutes les cellules par l'appel à `update()`

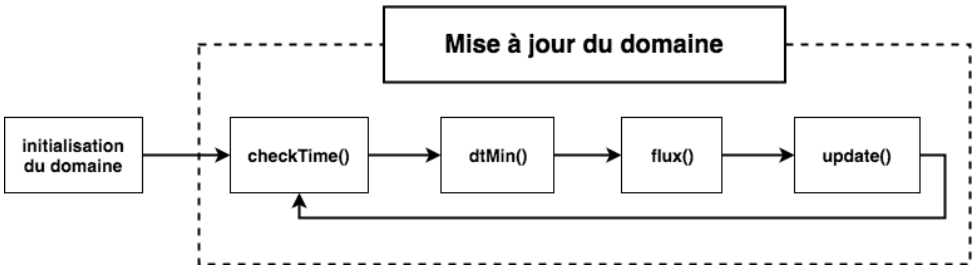


FIGURE 2.2 – Étapes d'exécution du programme

2.1.3 Benchmarking de l'exécution des fonctions

À partir de la structure d'exécution du code, il est possible de créer un profilage de l'exécution du programme. Celui-ci permet de mettre en évidence les parties du programme qui monopolisent le temps de calcul sur le CPU. Le profilage est un outil très utile pour disséquer et identifier le temps d'exécution d'un appel de fonction, les nombres d'appels à cette fonction et le pourcentage du temps de calcul que chacune des fonctions monopolise lors d'une étape. Par ailleurs, ces informations serviront à la section 2.1.5 pour calculer une limite théorique du ratio d'accélération par la parallélisation à l'aide de la loi d'Amdahl [15].

Le profiler utilisé est le programme **Instruments**, un logiciel propriétaire développé sur le système d'exploitation **macOS**. L'instrument de mesure proposé par le programme Instruments est **Time Profiler** [16]. Il capture les informations de la pile d'exécution à haut intervalle et peut ainsi enregistrer les informations d'exécution du programme SFlow2D s'effectuant sur un ou plusieurs processeurs.

Le tableau 2.1 regroupe uniquement les fonctions principales afin de ne pas entrer dans les détails d'implémentations et d'y afficher les appels les plus pertinents. Le tableau est réparti en 4 colonnes :

— La première colonne représente la structure du code et les appels imbriqués des fonctions sous forme d'arbre.

— La deuxième colonne est le temps total de toutes les exécutions d'une fonction.

— La troisième colonne illustre le pourcentage du temps total d'exécution du programme qu'une fonction ou une partie de l'exécution prend.

— La dernière colonne comptabilise pour une méthode, le nombre de fois qu'elle est exécutée.

Par exemple, le programme SFlow2D appelle la fonction `main`, elle-même appelant la mise à jour du calcul du domaine `CDomain:Update()` qui va lancer la mise à jour des cellules. L'ensemble des **85 174 320** appels à la méthode Update() sur chacune des **11 732** cellules effectuées sur les **7 260** itérations, prend un temps total de 20,81 secondes, soit 33,8 % du temps d'exécution.

Comme expliqué précédemment à partir de la figure 1.3a, le programme se décompose en 2 parties. La mise en place du domaine et le schéma itératif de mise à jour du domaine.

À partir du tableau 2.1, il est possible de tirer plusieurs conclusions sur le programme sans optimisation. Premièrement, l'initialisation du domaine est un processus rapide par rapport à la mise à jour du domaine. Ensuite, parmi toutes les méthodes, celles qui prennent le plus de temps sont le calcul des différents types de flux (flux de types Ci_RoeS0upwind, CiWall, CiHydrogram ou CiTransmissive dans le cas de Toce), la mise à jour des variables hydrauliques des cellules et le calcul du pas de temps minimal.

Appel Fonction	**Temps d'exécution**	**% du temps d'exécution**	**Nombre d'appels**
SFLOW2D	1.02 min	100.0%	1
|–**main**	1.02 min	99.4%	1
Initialisation du domaine			1
|– **CDomain(char*)**	95.10 ms	0.1%	1
| |– ReadInterfaces(char*)	39.00 ms	0.0%	1
| |– ReadCells(char*)	33.60 ms	0.0%	1
| |– CGauge(char*, CDomain*, double)	17.90 ms	0.0%	1
| |– ReadNodes(char*)	2.80 ms	0.0%	1
|			
Mise à jour du domaine de calcul			
|– **CDomain : Update()**	**1.02 min**	**99.2**%	1
| ***Ecriture des fichiers résultats***			**7260**
|– COutput : Write(char*, CCell*, int)	82.80 ms	0.1%	584
|– CPicture : CheckTime(CCell*, int)	85.90 ms	0.1%	54
|– CGauge : CheckTime()	4.30 ms	0.0%	530
|– CMonitorSection : CheckTime()	2.50 ms	0.0%	530
|– CDomain : Write_Umax()	12.50 ms	0.0 %	6
|			
| ***Calcul du pas de temps minimal***			
|– CDomain : dtMin()	**2.4 s**	**3.8**%	7260
|			
| ***Calcul des flux aux interfaces***			
|– Ci_RoeS0upwind : Flux()	**35.68 s**	**58.0%**	**126207840**
|– CiWall : Flux()	171.50 ms	0.2%	2526480
|– CiHydrogram : Flux()	40.40 ms	0.0%	116160
|– CiTransmissive : Flux()	37.20 ms	0.0%	464640
|			
| ***Mise à Jour des variables***			
| ***des cellules***			
|– CCell : Update()	**20.81 s**	**33.8%**	**85174320**

TABLE 2.1 – Temps d'exécution des fonctions principales appelées par le programme SFlow2D

En moyenne, le temps d'exécution d'un appel est minimal. En effet, un appel à la méthode `Flux()` prend en moyenne 0,276 μs, un appel à la méthode `Update()` prend en moyenne 0,244 μs et un appel à la méthode `dtMin()` 330 μs.

Une exécution indépendante de la méthode `Flux()` ou `Update()` est très rapide, mais dans leur globalité ces méthodes consomme du temps CPU. En effet, il faut les appeler à chaque itération autant de fois qu'il y a de cellules. Par conséquent, une piste d'amélioration serait de paralléliser ces différents appels en les répartissant sur les différents processeurs.

Temps d'exécutions

La table 2.2 fournit les temps d'exécutions sur les deux cas tests en fonction du niveau d'optimisation du compilateur. Ce point sera abordé plus en détail dans la section 4.3. Les temps sur le Toce ont été mesurés 10 fois, le tableau reprend la moyenne et l'écart type de ces mesures.

	-O0 [écart type]	-O3 [écart type]
Toce	219,1 s [1,78s]	135,9 s [2,77s]
Tous	84 h 8 min 1 s	54 h 0 min 38 s

TABLE 2.2 – Mesure du temps d'exécution : programme de base

La section suivante partira des conclusions tirées et les analysera plus en profondeur. Nous identifierons les éléments totalement parallélisables sans contraintes de synchronisation mais également les parties qui nécessitent plus d'attention pour éviter des conflits pouvant apparaître lors de la mise à jour de variables communes à deux exécutions parallèles.

2.1.4 Parties du programme parallélisables

Les trois parties principales du programme identifiées comme parallélisables sont le calcul du pas de temps minimal, le calcul du flux à toutes les interfaces et la mise à jour des variables hydrauliques de toutes les cellules.

Pas de temps minimal

Le calcul du pas de temps minimal se fait en calculant le pas de temps pour chacune des cellules et en gardant la valeur la plus petite avec un minimum prédéfini dans le cas où la hauteur de l'eau n'atteint pas un seuil minimal `"ZERO_DEPTH"` dans aucune d'elles.

Calcul du flux

Le calcul de flux à une interface nécessite la lecture d'informations des deux cellules voisines et la mise à jour à la fin du calcul d'une variable hydraulique dans chacune d'elles. Le problème peut donc se poser lorsque deux interfaces, appartenant à une même cellule, veulent modifier leur variable hydraulique en même temps.

Mise à jour des cellules

La mise à jour des variables hydrauliques des cellules se fait pour chaque cellule de manière indépendante. Une cellule ne met à jour que ses propres informations et ne requiert ainsi aucune attention pour éviter les conflits d'écriture. Il faut tout de même noter que la mise à jour du contenu d'une cellule ne peut se faire qu'à partir du moment où le calcul du flux à toutes ses interfaces a été effectué.

Écriture des fichiers de résultats

L'écriture de fichiers quant à elle, nécessite l'accès à toutes les informations concernant les cellules à un temps précis. Les itérations sont bloquées jusqu'à la fin de la génération des fichiers.

2.1.5 Limite théorique d'accélération

Dans un programme, certaines parties du code seront séquentielles et donc non parallélisables. Plus la proportion de ce type de parties est élevée, moins le code sera efficace en parallèle. La loi d'Amdahl [15] observe le phénomène d'amélioration du temps de calcul en fonction du programme :

$$S(s) = \frac{1}{1 - p + \frac{p}{s}} \tag{2.1}$$

— Le s est le nombre de cœurs de calcul.

— Le S indique l'accélération théorique du programme en fonction du s.

— Le p est la proportion du code en termes de temps d'exécution qui tire profit de l'exécution parallèle. Il s'agit donc de la proportion du code qui est parallélisable.
— Le $1 - p$ est la proportion du code en termes de temps d'exécution qui sera toujours séquentielle et ne pourra pas être parallélisée.

De plus, de cette loi découlent les propriétés suivantes :

$$S(s) \leq \frac{1}{1-p} \tag{2.2}$$

$$\lim_{s \to \infty} S(s) = \frac{1}{1-p} \tag{2.3}$$

La première équation donne la limite théorique que peut atteindre un programme en fonction des capacités matérielles ajoutées.
Les deux équations 2.2 et 2.3 montrent la limite de l'amélioration du temps. À partir d'un moment, il devient inutile d'ajouter plus de capacités matérielles, les performances vont plafonner.

La loi d'Amdahl [15] est utilisable en se basant sur les résultats obtenus dans la table 2.1 et en connaissant les parties du code parallélisable : le calcul du pas de temps minimal, le calcul du flux et la mise à jour des variables hydrauliques. En additionnant ces trois parties, la proportion parallélisable de l'exécution est de $95,8\%$. En se basant sur la loi d'Amdahl [15], en augmentant le nombre de cœurs du processeur utilisé de 1 à 2, le gain de temps serait de $1,92$. Et si le nombre de cœurs monte à 8, le temps d'exécution serait diminué de $6,18$ par rapport au temps initial ! Ces résultats sont théoriques et certaines parties ne sont pas entièrement parallélisables. Par exemple, le dtMin ou le calcul des flux, doivent accéder à des ressources communes, il faut donc les protéger ce qui augmente le temps d'exécution.

2.1.6 Calcul du gain et de l'efficacité

Le gain, quant à lui, permet de voir l'accélération d'un programme amélioré par rapport à sa version de base. Pour calculer celui-ci, la formule suivante est employée :

$$gain = \frac{Temps\ de\ calcul\ du\ programme\ de\ base}{Temps\ de\ calcul\ du\ programme\ amélioré}$$

L'efficacité permet d'analyser le gain d'un programme par rapport à son nombre de threads utilisés :

$$efficacité = \frac{gain}{\#\ threads}$$

L'efficacité donne une mesure de l'utilité de l'ajout de ressources.

2.2 Validation des résultats

Cette section traite de la problématique liée à la génération de résultats et propose une méthode de validation des améliorations.

2.2.1 Un problème de précision

Le programme SFlow2D est déterministe dans son exécution. Il calcule la mise à jour de son domaine de manière itérative et effectue des opérations dont l'ordre ne change, en principe, pas le résultat. La comparaison des instantanés de l'écoulement, en effectuant la boucle le calcul des flux aux interfaces dans un sens et ensuite dans l'autre, a cependant fait ressortir des problèmes de précision de calcul. Ces différences de précision qui n'impactent pas le début de la simulation prennent de plus en plus d'ampleur au fur et à mesure de l'avancement de la simulation. Une

approche statistique a été prise en vue de concevoir un protocole de validation.

L'analyse du programme non parallélisé a fait ressortir des différences de valeurs en fonction du sens dans lequel les interfaces sont parcourues. Le programme parallélisé ne respectera pas un ordre de lecture des interfaces, dès lors une différence existera toujours.

La figure 2.3 fait apparaître des différences significatives entre l'instantané 34 et l'instantané 35 lors de la comparaison du cas test Toce avec le programme de base en modifiant le sens de parcours. Les cellules, provoquant des différences importantes par rapport à la moyenne, sont les cellules qui sont proches d'une limite interne au domaine, en l'occurrence, un bloc en béton induisant de la traînée dans l'écoulement et une profondeur d'eau très faible. Un recensement de ces cellules a été représenté dans l'image 2.4.

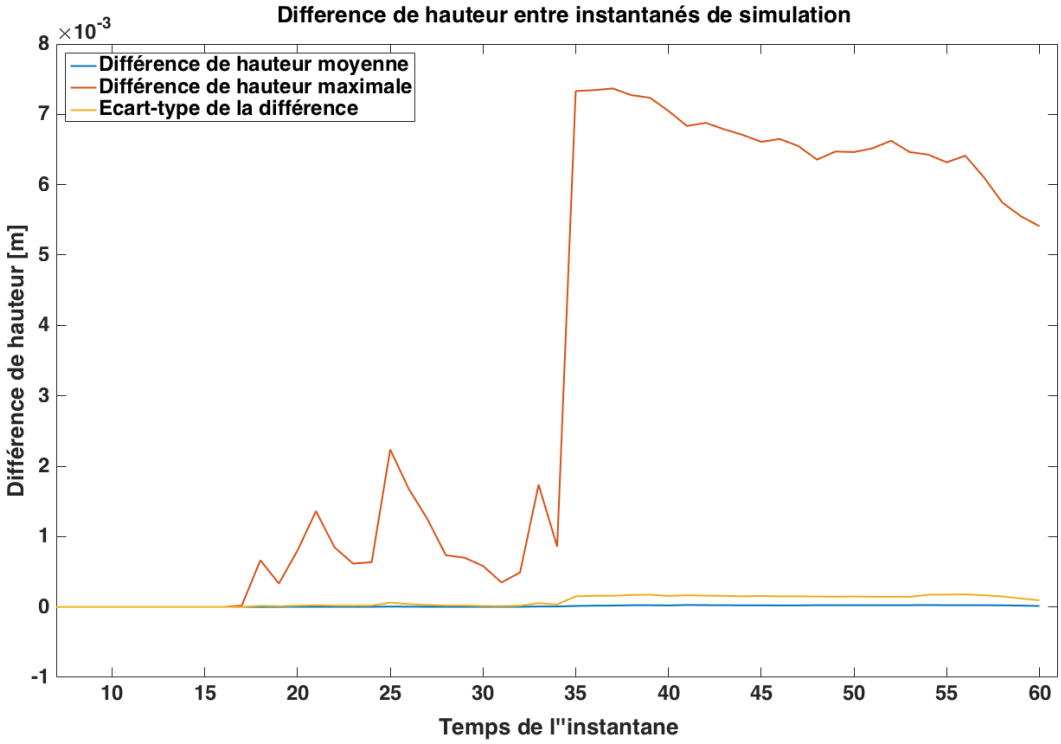


FIGURE 2.3 – Différence de hauteur entre instantanés de la simulation du cas test Toce

Les résultats de l'analyse statistique ont permis de définir une borne supérieure sur la différence de hauteur fixée à $8 \cdot 10^{-3}$m. Le même procédé a été appliqué pour la différence de débit par unité de longueur selon l'axe des x et selon l'axe des y entre 2 instantanés au même repère temporel. La borne supérieure pour le débit selon x et y est fixée à $4 \cdot 10^{-3}$ $\frac{m^2}{s}$.

En résumé, ce chapitre a déterminé que la parallélisation dans le cas de SFlow2D n'est pas triviale. Afin d'obtenir les meilleurs résultats, il faut mettre en place des mécanismes de synchronisation calcul du nouveau pas de temps, l'écriture de fichiers et le calcul de flux. Le chapitre suivant présente la séparation des cellules en régions, étape indispensable pour l'exécution sur un cluster. Le chapitre 4 se base ensuite sur l'analyse du programme pour introduire les modifications apportées permettant une exécution parallèle efficace.

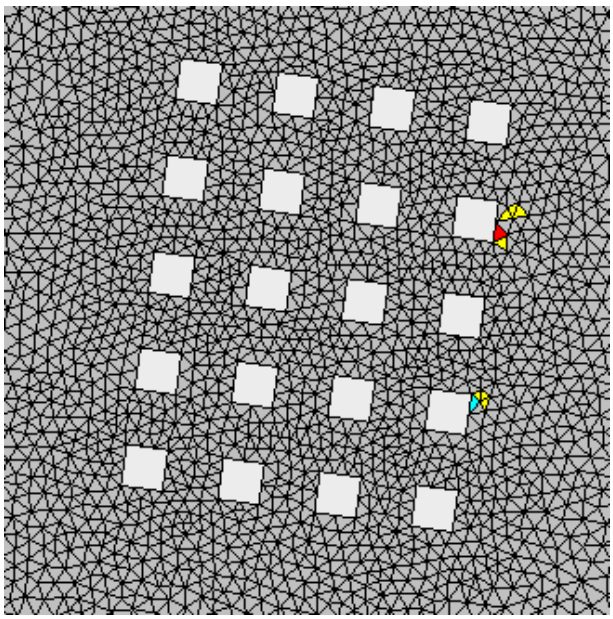


FIGURE 2.4 – Cellules affichant des différences de hauteur importantes

3
Division en régions

Un des enjeux de la parallélisation est la division du domaine de calcul en différentes régions dont les calculs seront envoyés sur les différents processeurs, ou cœurs de calcul. Chaque cœur de calcul sera responsable d'une région de cellules contiguës de façon à diminuer les échanges d'informations entre ces cœurs. Cependant, le problème de division du maillage en région de taille identique et de frontières entre régions minimales est un problème (NP-)difficile.

Des programmes libres tels que `METIS` [17] permettent de résoudre ce type de problème. Mais au vu de nos options de nos Masters respectifs (intelligence artificielle : données massives, optimisation et algorithmes), nous trouvions ce problème d'optimisation intéressant à résoudre et il formait une partie intégrante du travail à fournir. L'ensemble des programmes développés sont modulables, par exemple l'entrée du programme MPI a besoin d'un partitionnement en entrée, mais ce dernier peut être généré par notre programme de division en région ou venir de toute autre source.

Plusieurs points sont importants pour veiller au bon fonctionnement du calcul du flux d'eau du point de vue de la division :

— La taille des régions : Chaque région doit posséder un nombre similaire de cellules. Si une région possède plus de cellules que la moyenne des autres régions, le nœud du cluster qui sera chargé de cette région prendra plus de temps que les autres et risquera de retarder tous les autres nœuds. Un pourcentage par rapport à la moyenne du nombre de cellules par région permet de créer une borne inférieure et supérieure acceptable pour la taille d'une région.

— La taille des frontières : Le nombre de cellules présentes à la frontière d'une région représente la quantité d'information à faire passer entre deux nœuds du cluster. Il faut donc minimiser la taille des frontières entre les différentes régions pour limiter le temps des passages d'information.

— La rapidité : La division doit être rapide par rapport au temps de calcul total. Si un cluster est employé, c'est pour augmenter significativement la vitesse de calcul, et si la division en régions prend un temps important, il vaut mieux lancer la simulation sur un seul ordinateur. Il faut donc que le temps nécessaire pour la division additionné à celui du calcul sur cluster soit inférieur au temps de calcul sur un unique ordinateur.

Ce chapitre présente tout d'abord l'équivalence du problème de partitionnement optimal du domaine au problème de partitionnement optimal du graphe et une preuve succincte de la NP-difficulté de ces deux problèmes. Ensuite, deux pistes seront explorées pour trouver une solution au problème de division. La première approche est l'emploi d'un solveur pour trouver la solution exacte. La seconde approche consiste à trouver une solution initiale déséquilibrée, à échanger des cellules entre les régions pour les équilibrer et finalement y appliquer un algorithme de recherche pour trouver la frontière optimale. Enfin, une dernière section sera consacrée à la discussion détaillée des résultats.

3.1 Notations et propriétés

Cette première section présente les notations et les propriétés utilisées par la suite dans le but de construire une définition mathématique du problème.

Dans le domaine D (**C**, **I**) où **C** est l'ensemble des *cellules* du domaine et **I** l'ensemble des *interfaces* du domaine, il convient de définir les notions suivantes :

- Une **interface** $i_{k,l} \in \mathbf{I}$ est la frontière commune à 2 cellules adjacentes (k, l)
- Un **sous-domaine** du domaine D (C, I) est un domaine (C', I') avec C' $\subseteq$ C et I' $\subseteq$ I.
- Un **parcours** est une suite $c_0\ i_1\ c_1\ i_2\ c_2\ ...\ i_n\ c_n$ où c_0, c_1, ... sont des cellules et i_1, i_2,... sont des interfaces.
  - La longueur du parcours est son nombre d'interfaces n.
  - La cellule d'*origine* est c_0 et la cellule de *destination* est c_n.
  - Les autres cellules sont dites *intérieures*.
- Un **chemin** est un parcours dont les cellules et les interfaces sont toutes distinctes.
- Un domaine est **connexe** si pour chaque paire de cellules il existe un parcours qui les relie.

Propriété 3.1 *Une interface est dite frontière si les 2 cellules qui lui sont adjacentes appartiennent à des sous-domaines différents.*

Le chemin le plus court entre 2 cellules consiste à trouver le nombre minimal d'interfaces permettant de relier 2 cellules.

Soit un domaine D, et une fonction de poids unitaire $f : I \to 1$, le chemin le plus court ou *shortestPath(o, d)*, d'*origine o* et de *destination d*, est défini par :

$$shortestPath(o, d) = \min_{k \in 1..n} \sum_{j=0}^{k} f(i_{j,j+1})$$

3.2 Le problème de partitionnement

Le problème de partitionnement est un problème d'optimisation qui vise à uniformiser la répartition des cellules dans les différents sous-domaines et à minimiser la taille de la frontière de chaque sous-domaine. Cette section formalise le problème de partitionnement du domaine, lie celui-ci au problème de partitionnement de graphes et explique pourquoi il est si difficile de trouver la solution optimale à ces deux problèmes.

3.2.1 Le partitionnement du domaine de calcul

Afin de s'assurer de l'exactitude d'un partitionnement P sur le domaine D (C, I), il est important que celui-ci possède les propriétés 3.2, 3.3, 3.4, et 3.5

Propriété 3.2 *L'union des ensembles de cellules C_i des sous-domaines D_i correspond à l'ensemble des cellules C du domaine initial D(C, I)*

$$C = \bigcup_{i=1}^{n_r} C_i$$

Propriété 3.3 *Une cellule c_i n'appartient qu'à un seul ensemble de cellules C_i d'un sous-domaine D_i du domaine initial D(C, I).*

$$\forall i, j \in 1..n_r \text{ où } i \neq j\ \forall c_i \in C_i, c_j \in C_j : c_i \neq c_j$$

Propriété 3.4 *Les cellules C sont réparties de manière équilibrée à travers les différents sous-domaines D_i.*

$$\forall i \in 1..n_r : lb \leq |C_i| \leq ub$$

Cette dernière équation offre une marge au partitionnement, définie par une borne inférieure lb et une borne supérieure ub sur le nombre de cellules $|C_i|$ présentes au sein d'un même sous-domaine D_i.

Propriété 3.5 *Il existe un chemin entre toutes les cellules d'un sous-domaine.*

Les propriétés ci-dessus permettent donc de formaliser le problème du partitionnement qui prend la forme de l'équation ci-dessous

$$\min_{c \in C} \sum_{i=1}^{n_r} (|C_i| - \mu) + \sum_{j=1}^{|C_i|} front(c_j) \text{ où } \forall i \in 1..n_r : lb \leq |C_i| \leq ub \tag{3.1}$$

où la fonction $front(c_j)$ = nombre d'interfaces frontières de la cellule c_j

3.2.2 Un problème équivalent

Une des démarches pour démontrer que la division optimale revient à résoudre un problème NP-difficile, est de trouver un problème équivalent, dont la preuve de la NP-difficulté a déjà été faite. Le partitionnement optimal de graphe est un problème prouvé NP-difficile.

Un graphe se définit comme un triplet (V, E, ϕ) où
- **V** est un ensemble fini dont les éléments sont appelés sommets ou nœuds
- **E** est un ensemble fini dont les éléments sont appelés arêtes.
- ϕ est une fonction, dite fonction d'incidence, qui associe à chaque arête un sommet ou une paire de sommets.

Il est donc possible de construire un graphe complémentaire à partir du domaine de calcul. Les centres des cellules du domaine de calcul sont les nœuds du graphe et il existe une arête entre 2 nœuds si 2 cellules sont adjacentes. La fonction d'incidence est analogue aux interfaces.

De plus, les propriétés qui sont définies sur le domaine sont également valables sur le graphe complémentaire. La notion de sous-domaine se traduit donc par la notion de sous-graphe.

Propriété 3.6 *Un sous-graphe du graphe (V, E, ϕ) est un graphe (V', E', ϕ) avec $V' \subseteq V$, $E' \subseteq E$, ϕ' est la restriction de ϕ à E'.*

Identiquement les 2 conditions sur le partitionnement se traduisent de la façon suivante :

Propriété 3.7 *Un sommet ne peut appartenir qu'à un seul sous-graphe*

$$\forall i, j \in 1..n_r \text{ où } i \neq j \ \forall v_i \in V_i, v_j \in V_j : v_i \neq v_j$$

Propriété 3.8 *Les cellules sont réparties de manières équilibrées dans les sous-graphes*

$$\forall i \in 1..n_r : lb \leq |V_i| \leq ub$$

Propriété 3.9 *L'union de tous les sommets V_i des sous-graphes G_i correspond à l'ensemble des sommets V graphe initial G(V, E).*

$$V = \bigcup_{i=1}^{n_r} V_i$$

Ainsi le problème de partitionnement du domaine est équivalent à trouver le partitionnement du graphe G(V, E) satisfaisant les propriétés 3.7, 3.8 et 3.9.

3.2.3 Un problème NP-difficile

En calculabilité [18], un problème dans la classe NP, est un problème pour lequel, il n'existe pas de limite de temps garantie décrite par un polynôme dont la variable serait la taille de l'entrée. La classe des problèmes NP-difficiles regroupe les problèmes de décision qui sont au moins aussi difficiles que les problèmes les plus difficiles dans NP (mais n'appartiennent pas forcément à la classe des problèmes NP). Enfin, la classe des problèmes NP-complets contient les problèmes plus difficiles dans NP. Chaque problème NP-complet doit être dans NP.

Intuitivement, le problème de regrouper les nœuds d'un graphe en régions de tailles égales peut paraître facile, mais il est connu comme un problème NP-difficile. Dyer and Frieze [19] fournissent une preuve de cette affirmation. Par conséquent, le problème de partitionnement du domaine en régions de tailles égales de frontières minimales est un problème NP-difficile. Les solutions au problème de partitionnement, comme dans le cadre de ce mémoire, sont souvent approximées à partir d'heuristiques.

Les prochaines sections présenteront deux approches en vue de trouver le meilleur partitionnement se rapprochant au mieux du partitionnement idéal.

3.3 Gurobi Optimizer

Une première approche permettant la résolution du problème de partitionnement est l'utilisation de la programmation par contraintes couplée à un solveur, et ce, afin d'explorer l'ensemble des solutions possibles pour trouver la solution optimale.

La programmation par contraintes permet de définir un modèle, en général une fonction d'objectif à optimiser avec ses variables et d'y ajouter des contraintes. Le but des contraintes est double : contraindre les variables et restreindre le champ d'exploration des solutions. Cette section présente cette approche afin de trouver la solution optimale au problème de division.

Le solveur employé est Gurobi Optimizer [20, 21]. Il est l'un des solveurs les plus rapides et puissants pour résoudre des problèmes tels que des programmes linéaires (LP), quadratiques et d'entier mixte (MIP). Il est employé par plus de 1 600 entreprises et offre gracieusement des licences complètes pour les étudiants ou membres universitaires. De plus, il supporte de nombreux langages de programmation et plates-formes.

Pour créer le modèle à résoudre sur le solveur, il faut définir ses variables et ses contraintes. Dans le cas du partitionnement optimal, les variables sont :

- Des couples sous la forme (c, r), avec c le numéro d'une cellule du domaine (par exemple un triangle), et r un numéro d'une région. Cette variable est binaire, elle peut donc valoir 1 si ce couple existe, donc celle cellule fait partie de cette région ou sinon 0. S'il y a N cellules et R régions, il y aura $N * R$ variables de ce type-là.
- Des couples sous la forme $(c1, c2)$ qui lient deux cellules voisines. Elles sont également du type binaire et valent 1 si les deux cellules appartiennent à des régions différentes ou sinon 0. Il y a autant de ces variables que de frontières voisine à deux cellules.

Différentes contraintes permettent de fixer les variables :

- Des contraintes qui fixent la variable de deux cellules voisines à 0 si elles font partie de la même région. Il y a 2 fois plus de contraintes de ce type que le nombre de frontières voisines à deux cellules.
- Des contraintes qui vérifient que chaque cellule n'appartient qu'à une seule région. Il y a autant de ces contraintes que de cellules.

— La dernière contrainte s'assure que la différence entre deux tailles de région soit inférieure à une certaine borne. Il y a donc nombre_de_régions^2 contraintes de ce dernier type.

La fonction objectif est la taille des frontières, cela veut dire que la taille totale des frontières minimales est recherchée. Une frontière est définie comme une interface qui se trouve entre deux cellules qui n'appartiennent pas à la même région.

3.3.1 Résultats

Le but du solveur est de trouver la solution optimale et de prouver que celle-ci est l'optimum. Pour ce faire, il doit parcourir tout l'arbre de recherche pour prouver qu'il n'y a pas de meilleure solution. Il emploie des techniques pour diminuer la taille de cet arbre en fonction des résultats trouvés précédemment.

Cependant, le solveur prend un temps énorme pour finir sa recherche et donc prouver l'optimum. Un maillage relativement simple a été employé pour démontrer que le solveur trouvera la solution optimale au profit de la vitesse. Celui-ci est composé de 100 triangles, ils sont rangés sur une grille de 10 sur 5 et chaque carré de la grille est coupé en deux sur sa diagonale montante. Pour prouver l'optimum, il prend plus de 12 h en fonctionnant sur 4 cœurs d'un ordinateur ! Par contre, il trouve des solutions temporaires en un temps plus restreint :

— Il atteint une frontière de 16 au bout de 75 secondes. Cela signifie donc qu'il a trouvé une division de cellules dont le nombre d'interfaces communes à 2 régions est de 16.

— Il obtient une frontière de 15 après 41 minutes. La valeur obtenue est optimale, mais le solveur ne le sait pas encore, il doit encore continuer sa recherche pendant une dizaine d'heures avant de prouver l'optimalité.

Il est possible d'ajouter une limite de temps à l'optimisation ou une limite entre la borne inférieure et la supérieure de la valeur objectif, mais il est difficile d'estimer correctement ces paramètres à l'avance puisqu'ils peuvent varier fortement en fonction du maillage du domaine et du nombre de régions attendu.
De plus, le solveur ne trouve pas uniquement des régions contiguës, donc s'il a une limite de temps, la réponse retournée ne respectera peut-être pas la propriété 3.5. Par contre, si le solveur trouve l'optimum, chaque région respectera cette propriété.

Le code du solveur a été réalisé avec l'aide de Guillaume Derval, assistant-doctorant à l'Université Catholique de Louvain. Il est toutefois possible d'améliorer ce code, mais cela sort du cadre de notre mémoire au vu de la complexité de Gurobi et l'objectif de notre travail.

3.4 Méthode approchée

La seconde approche est de trouver en un temps assez court une solution correcte par une méthode approchée séparée en 4 étapes.

Les 4 grandes étapes comme représentées sur la figure 3.1 sont : le calcul de la matrice des distances (si nécessaire), la recherche des centroïdes, l'attribution de chaque cellule à un centroïde et l'amélioration du partitionnement initial. L'heuristique de recherche va exécuter les étapes dans l'ordre pour ne garder que le meilleur résultat en se basant sur différentes initialisations des centroïdes.

3.4.1 Matrice des distances

Pour représenter efficacement une région avec un maillage, les formes employées n'auront pas toutes des dimensions similaires et formeront donc un maillage irrégulier. Par exemple, il est utile d'avoir des petits triangles dans les zones proches des bordures pour avoir plus de précision

et des triangles plus grands pour les zones plus éloignées [22].

Pour la suite, nous aurons besoin de connaître la cellule la plus proche d'un certain centroïde. Il aurait été facile de pouvoir calculer cette distance en employant les coordonnées des deux centres. Mais à cause des différences de taille de cellule et du maillage irrégulier, il faut calculer la distance entre chaque cellule en comptant le nombre minimal de cellules qui les séparent.

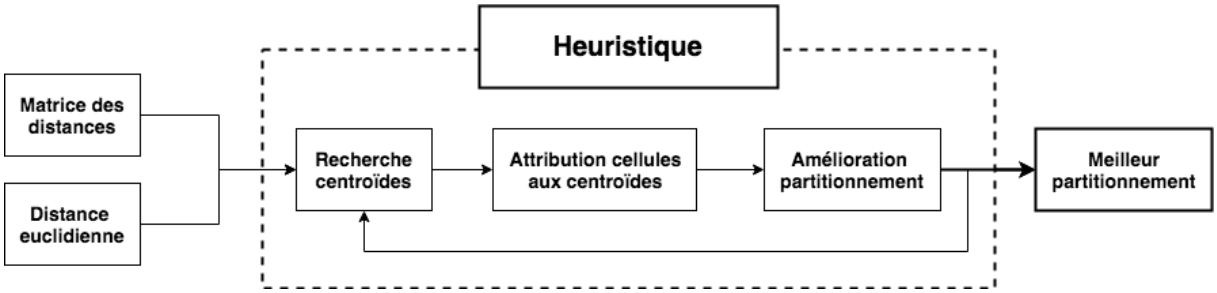


FIGURE 3.1 – Étapes d'exécution de la méthode approchée

Pour calculer ces distances, une version modifiée de l'algorithme de Floyd-Warshall [23] a été employée.

Le nouvel algorithme de calcul des distances est représenté par le pseudocode `Algorithme` 1. Dans cet algorithme, la liste S contient tous les tuples composés d'une origine et d'une destination et fonctionne de la manière suivante : Lorsque le premier élément de la liste (qui a donc été ajouté le premier) est traité, la cellule de destination annonce à tous ses voisins qu'ils peuvent atteindre la cellule de départ en une certaine distance. Si un voisin n'a pas encore de distance connue, il va l'enregistrer et la distance sera ajoutée à la pile pour qu'il puisse à son tour l'annoncer à tous ses voisins. Si la distance est déjà connue, il va l'ignorer et ne la transmettra pas.

Algorithme 1: Calcul de la matrice des distances

Data:
Cell ⟵ liste des cellules
Neighbors[c] ⟵ les voisins de chaque cellule
Result: D : |Cell| x |Cell|, la matrice des distances
initialisation de la matrice des distances à -1
$S \longleftarrow \emptyset$ `/* Création de la liste des distances en attente d'être traitées */`
for $c \in$ *Cell* **do** `/* chaque nœud peut s'atteindre avec une distance de 0 */`
  $D_{c,c} \longleftarrow 0$
  S.pushBack((c,c))
while $S \neq \emptyset$ **do**
  ($dest$, $source$) ⟵ S.pop()
  for $n \in$ *Neighbors[dest]* **do**
    if $D_{source,n} == -1$ **then** `/* Ce voisin de dest n'a pas encore de distance pour la cellule source */`
      $D_{source,n} = D_{source,dest} + 1$
      S.pushBack($(n,\ source)$) `/* ajout à la fin de la liste */`

La propagation se fera donc comme une onde. Si une distance entre deux nœuds n'existe pas encore, la première valeur à arriver sera la plus courte. Cette affirmation est correcte puisque la distance entre deux nœuds voisins vaut toujours 1 et qu'une nouvelle distance à traiter est

insérée à la fin de la liste et donc traitée après les distances insérées antérieurement.

Cette première partie de la méthode approchée prend un temps relativement important et les distances calculées seront identiques pour une même topologie, peu importe le nombre de régions désirées. Elle occupe également une grande quantité de RAM. Les distances peuvent donc être enregistrées dans un fichier pour ensuite être fournies en entrée.

De plus, la taille des fichiers peut être un problème à long terme. Pour diminuer la taille de ceux-ci, il y a un peu moins de la moitié des données qui sont enregistrées. En effet notre problème étant symétrique : la distance de A vers B est identique à celle de B vers A, nous n'enregistrons que les valeurs pour lesquelles le nombre de la destination est supérieur à celui de l'origine. De plus, nous n'enregistrons pas la diagonale descendante de la matrice puisqu'elle correspond à la distance de A vers A, qui est toujours nulle.

Grâce à ces deux améliorations, nous pouvons diminuer par un facteur deux la taille du fichier. Cette diminution est également bénéfique à la mémoire RAM lors de l'exécution de la suite de la méthode approchée.

Malheureusement, la RAM de l'ordinateur nécessaire au calcul est trop importante pour de très grandes simulations. Par exemple, avec la simulation de la rupture du barrage de Tous [3], la mémoire RAM nécessaire est supérieure à 14 Gb dû à son nombre élevé de cellules. Pour cette raison, il est également possible d'éviter l'usage de la matrice des distances. À la place, le programme utilise la distance euclidienne entre les deux centres des cellules. C'est un gain de temps important au début de l'exécution puisqu'il ne faut pas calculer la matrice des distances. Par contre, pour chaque recherche de distance, il faut la recalculer en se basant sur les coordonnées. La qualité de la solution est également diminuée à cause de la taille inégale des formes qui composent le maillage et du mailage même.

3.4.2 Centroïdes

L'étape suivante consiste à trouver les points de départ pour faire générer les régions. En outre, il en faut autant que le nombre de régions souhaitées. Pour obtenir des résultats intéressants dans la suite de la méthode approchée, il faut trouver des centroïdes les plus éloignés les uns des autres.

L'algorithme de calcul des centroïdes est simple et efficace, il utilise la matrice des distances calculée précédemment (ou la distance euclidienne si cette dernière est inexistante).

Pour commencer, il sélectionne aléatoirement autant de cellules que de régions, ce sont les N centroïdes de départ. Ensuite, il prend chaque centroïde et va le remplacer par une nouvelle cellule. Pour sélectionner la nouvelle cellule, il va toutes les parcourir et sélectionner celle qui aura la distance maximale en considérant seulement le centroïde le plus proche. En langage formel :

$$\text{nouveauCentroïde} = \max_{\text{cell} \in \text{Cell}} \left(\min_{\text{centroïde} \in \text{Centroïde}} \Big(\text{dist[cell][centroïde]} \Big) \right) \tag{3.2}$$

Dans l'équation 3.2, dans l'ensemble des centroïdes, il ne faut pas inclure celui qui est en train d'être déplacé. Si aucun des centroïdes n'a été déplacé lors de l'itération, cela signifie qu'aucun d'eux ne peut trouver une meilleure position par rapport aux autres, ce qui signifie la fin de cette étape.

Cet algorithme est semblable à un autre algorithme connu sous le nom de *k-means*. Cet algorithme consiste à assigner à un nuage de points k centroïdes, puis attribuer les cellules au centroïde le plus proche formant ainsi des régions. Ensuite, l'algorithme calcule la nouvelle position des centroïdes dans l'espace en trouvant le barycentre de chaque région. Il répète l'exécution en réattribuant les cellules au centroïde le plus proche et en recalculant le barycentre jusqu'à atteindre un équilibre. L'algorithme 2 n'utilise pas le barycentre, mais la maximisation de la distance minimale entre les centroïdes et toutes les cellules pour permettre de trouver les centres d'un nombre équitable de cellules.

Algorithme 2: Calcul des centroïdes

```
Data: Cell ⟵ liste des cellules
Result: Centroid : liste des cellules centroïdes
Centroid ⟵ { }
for i ∈ 1 .. #centroids do
    Centroid.pushBack(randomCell())
changement ⟵ true while changement do
    changement ⟵ false
    for centroid ∈ Centroid do
        Centroid.remove(centroid)
        minDist ⟵ MAX_VALUE
        for c ∈ Cell do
            if MinDistWithAllCentroid(c) < minDist then
                minDist ⟵ MinDistWithAllCentroid(c)
                newCentroid ⟵ c
        if newCentroid ≠ centroid then
            changement ⟵ true
        Centroid.pushBack(newCentroid)
```

3.4.3 Attribution initiale des cellules

La troisième étape de la méthode approchée répartit les cellules entre les centroïdes pour former des régions. Cette répartition doit être rapide et doit distribuer de manière relativement équitable les cellules entre les centroïdes.

Chaque région peut choisir et acquérir à son tour une nouvelle cellule disponible, jusqu'à ce qu'il n'y ait plus de cellule voisine libre. La région choisit chaque fois la cellule qui est la plus proche de son centroïde. Dans certaines situations, une région se fait bloquer par d'autres régions qui ont déjà choisi des cellules proches d'elle, la première région ne sait donc plus grandir puisqu'elle n'a plus de cellules disponibles. À l'opposé, certaines régions ont beaucoup de cellules disponibles à côté d'elles et donc grandiront fortement.

Si cette fonction réalise correctement son travail, les prochaines étapes seront plus légères à exécuter puisqu'il faudra moins déplacer de cellules des régions trop grandes vers celles trop petites. La figure 3.4a représente le résultat de l'attribution initiale des cellules.

3.4.4 Amélioration de l'attribution des cellules

La dernière étape est l'amélioration de la division en régions. Cette étape est la plus importante. Celle-ci modifie l'attribution des cellules pour modifier la taille des régions qui ne sont pas

dans les bornes définies comme acceptables.

Un premier algorithme, l'algorithme 3, a été développé pour facilement vérifier que, malgré le déplacement d'une cellule d'une région à l'autre, le domaine reste contigu.
Le second algorithme, l'algorithme 4, sert à proprement parler à améliorer la répartition des cellules. Celui-ci utilise intensivement le premier algorithme.

Cellules contiguës

Il est important qu'une région reste contiguë. Il faut donc veiller, à chaque nouvelle obtention d'une cellule par une région, qu'elle ne coupe pas une autre région.

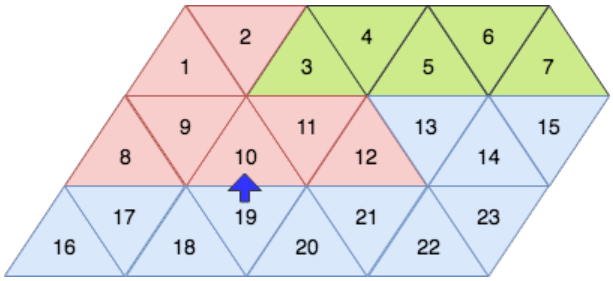


FIGURE 3.2 – Domaine avec trois régions

Par exemple, dans la figure 3.2, ni la cellule 10 ni la 11 ne peuvent être prises à la région rouge au risque de couper son domaine.

Algorithme 3: Recherche l'existence d'un chemin entre deux voisins

Data: *c1* (voisin 1), *c2* (voisin 2), *cellRemoved* (cellule qui va être retirée et qui possède les deux voisins)
Result: *true* si un chemin existe, *false* ou sinon
listCells1 : set des cellules visitées depuis c1
listCells2 : set des cellules visitées depuis c2
listCellsToExtend1 : liste des cellules à étendre depuis c1
listCellsToExtend2 : liste des cellules à étendre depuis c2
listCells1.pushBack(*c1*)
listCells2.pushBack(*c2*)
listCellsToExtend1.pushBack(*c1*)
listCellsToExtend2.pushBack(*c2*)
while *listCellsToExtend1* $\neq \emptyset$ *and listCellsToExtend2* $\neq \emptyset$ **do**
  c1 $\longleftarrow$ *listCellsToExtend1*.popFront()
  c2 $\longleftarrow$ *listCellsToExtend2*.popFront()
  if *c1.extend* $\in$ *listCells2* **then**
    return true
  if *c2.extend* $\in$ *listCells1* **then**
    return true
  listCells1.pushBack(*c1*.extend)
  listCells2.pushBack(*c2*.extend)
  listCellsToExtend1.pushBack(*c1*.extend)
  listCellsToExtend2.pushBack(*c2*.extend)
return false

Pour détecter ce genre de situations, un algorithme cherchant un chemin entre deux cellules a été développé. En effet, si une cellule d'une région est prise, sa frontière va être rognée, et il faut s'assurer qu'une nouvelle frontière contiguë pourra être formée. Prenons un exemple :

si la cellule qui va être enlevée possède deux voisins qui font partie de sa région, il faut vérifier qu'il existe un chemin qui relie ces deux voisins en ne passant que par des cellules de sa région.

Dans la figure 3.2, la région bleue tente de prendre la cellule 10, pour détecter une coupure, il faut vérifier qu'il existe un chemin qui passe dans la région rouge pour rejoindre la cellule 9 à la cellule 11. Un tel chemin n'existe pas, ce qui signifie que si cette action est exécutée, le domaine rouge deviendra discontinu.

Pour que l'algorithme 3 soit le plus efficace possible, un double arbre de recherche est employé. Cette efficacité est primordiale au vu du nombre d'appels à cette fonction, c'est-à-dire à chaque déplacement d'une cellule d'une région vers une autre. Avec ce système, un arbre de recherche est créé sur chaque nœud, ensuite les deux arbres grandissent en enregistrant tous les nœuds obtenus. Si en grandissant un arbre, un nœud qui est déjà présent dans la liste des nœuds du second arbre, est obtenu, il est certain qu'il existe un chemin pour relier ces deux cellules de départ. Ce système de double arbre est beaucoup plus efficace en comparaison avec la création d'un unique arbre qui démarrerait à un nœud et qui s'arrêterait quand il atteint le second point. Pour éviter de grandir inutilement, si un des deux arbres ne sait plus grandir, la recherche est arrêtée : il n'existe pas de chemin liant les deux cellules.

Dans l'algorithme 3, la fonction `extend` retourne tous les voisins d'une cellule qui font partie de la même région. Et un nœud n'est ajouté à une liste que s'il ne fait pas déjà partie de cette liste.

Algorithme d'amélioration du partitionnement

Pour améliorer la répartition des cellules dans une région, l'algorithme 4 est employé. Pour fonctionner, il a besoin de connaître toutes les informations des cellules (voisins de chaque cellule, position...), la répartition actuelle des cellules et la liste des centroïdes.
Les bornes inférieure et supérieure ont été définies comme une marge autour de la moyenne. Une région qui se situe entre ces deux bornes est considérée comme idéale. Pour éviter qu'une région ait trop de cellules, la moyenne est employée à la place de la borne supérieure dans l'algorithme. Cela favorise le déplacement des cellules d'une région à l'autre.

Algorithme 4: Amélioration de la distribution initiale des cellules

Data: *centroïdes* (liste des centroïdes)
c.regionID (numéro de la région de la cellule c)
N (le nombre de régions)
Result: regionID des cellules est modifié
region_cnt[N] : tableau du nombre de cellules par région
borderCell[N] : liste des cellules frontière de chaque région
$mean \longleftarrow \frac{numCells}{numRegions}$
player $\longleftarrow$ 0
while *solutionImproved* **do**
  if *region_cnt[player]* > *mean* **then**
    give_c, *recv_p* $\longleftarrow$ findCellToGive(*borderCells*, *centroids*, *player*) ;
    UpdatePartition(*give_c, player, recv_p*)
  else if *region_cnt[player]* $\leq$ *lowerBound* **then**
    give_c, *give_p* $\longleftarrow$ findCellToTake(*borderCells*, *centroids*, *player*)
    UpdatePartition(*give_c*, *give_p, player*)
  player $\longleftarrow$ (*player*+1) mod *numRegion*

La première étape consiste à calculer le nombre de cellules présentes dans chaque région ainsi que leurs frontières. Ensuite, chaque région à son tour va améliorer la répartition.

Si la région a plus de cellules que la moyenne, elle va sélectionner une de ses cellules qui est sur sa frontière pour l'offrir à une région possédant moins de cellules que la moyenne en appelant la `findCellToGive` représentée par l'algorithme 5. Pour trouver la meilleure cellule à donner, elle va calculer la différence entre la distance de son centroïde et de la cellule en question et entre la distance du centroïde de la région voisine avec cette même cellule. Ce calcul permet de trouver la cellule qui est la plus éloignée du donneur et donc qui l'arrange le moins, et tout en étant proche du centre de la région du receveur.

Algorithme 5: `findCellToGive(borderCells, centroids, player)`

```
Data: borderCells (liste de cellules frontières)
      centroïdes (liste des cellules centroïdes)
      player (joueur actuel)
Result: give_c (cellule à donner), recv_p (joueur recevant la cellule)
give_c ⟵ -1
recv_p ⟵ -1
for bCell ∈ borderCells[player] do
    for n ∈ Neighbour(bCell) do
        rID ⟵ n.regionID
        if region_cnt[rID] > mean then
            continue
        diff ⟵ ‖centroids[player] - bCell‖ - ‖centroids[rID] - bCell‖
        if diff.isMaxMove() then
            give_c ⟵ bCell
            recv_p ⟵ rID
return give_c, recv_p
```

Lorsque la cellule est sélectionnée, il faut vérifier que le don de cette cellule ne va pas couper le domaine du donneur. Cela est réalisé grâce à l'algorithme 6 et de l'appel à l'algorithme 3 explicité précédemment. Si le domaine n'est pas en danger de coupure, il calcule pour vérifier que le score s'améliore. Si c'est effectivement le cas, il procède au don.

Algorithme 6: `UpdatePartition(give_c, give_p, recv_p)`

```
Data: give_c (cellule à donner)        give_p (joueur donnant la cellule)
      recv_p (joueur recevant la cellule)
Result: mise à jour de la liste des cellules frontières, du nombre de cellules par région et
        de la liste des cellules frontières
if HavePath(give_c, neighbours(give_c), recv_p) then
    if scoreImproved() then
        GiveCell(give_c, give_p, recv_p)
        updateBorderCells()
        region_cnt[recv_p]++
        region_cnt[give_p]–
```

La seconde partie de l'algorithme 4 fonctionne de manière similaire, sauf que la région en manque de cellules, va parcourir toutes les cellules voisines à sa frontière pour voir si elle peut la lui prendre en appelant la `findCellToTake` représentée par l'algorithme 7. Avant le vol d'une cellule, il vérifie que la région cible a plus de cellules que la borne inférieure. Pour sélectionner la

meilleure cellule à prendre, il trouve le minimum des deux distances. Ce qui permet de sélectionner la cellule la plus intéressante pour la région en manque tout en étant le moins dérangeant pour la région qui va perdre une cellule.

Algorithme 7: `findCellToTake(borderCells, centroids, player)`

Data: *borderCells* (liste de cellules frontière)
centroïdes (liste des cellules centroïdes)
player (joueur actuel)
Result: *give_c* (cellule à donner), *give_p* (joueur donnant la cellule)

```
give_c ⟵ -1
give_p ⟵ -1
for bCell ∈ borderCells[player] do
    for n ∈ Neighbour(bCell) do
        rID ⟵ n.regionID
        if region_cnt[rID] ≤ lowerBound then
            continue
        diff ⟵ ||centroids[player] - n|| - ||centroids[rID] - n||
        if diff.isMinMove() then
            give_c ⟵ n
            give_p ⟵ rID
return give_c, recv_p
```

3.4.5 Recherche du meilleur partitionnement

La recherche du meilleur partitionnement des cellules est repris dans une fonction représentée sur la figure 3.1. Le principe est simple : L'heuristique calcule tout d'abord les centroïdes, effectue un partitionnement initial et améliore ce partitionnement. Ensuite, elle recommence ces trois étapes en sélectionnant des centroïdes différents. Si le score est amélioré par rapport à la meilleure itération jusqu'à présent, il l'enregistre ou sinon il recommence son itération. Pour définir le score, la somme des différences entre la moyenne et le nombre réellement présent dans chaque région est employée. Cette boucle est exécutée un nombre de fois prédéfinie en fonction de la distance employée. Si la distance euclidienne est employée, le domaine est supposé très grand, le nombre d'itérations ne sera pas trop élevé (5 itérations) pour obtenir un résultat en un temps raisonnable. Par contre avec la distance en termes de voisins, un résultat plus précis est attendu : 20 itérations seront effectuées.

Lorsque les itérations sont terminées, une seconde étape commence : l'échange de cellules (Swap en anglais) représenté par la figure 3.3. Cette seconde étape correspond à faire une *recherche locale*. La recherche est dite locale, car lors de cette étape, les régions vont s'échanger leurs cellules frontières, pour atteindre un minimum local : un partitionnement dont le score est minimal. Le nombre de cellules par région va donc rester strictement identique. Un nouveau type de score va être employé : la taille des frontières. Cette étape permet de diminuer le nombre de cellules frontières. Tant que le score peut être amélioré, l'exécution ou autrement dit, la recherche locale continue. Il cherche pour tous les couples de régions possibles (*Région1*, *Région2*) si la Région1 sait donner une cellule frontière à la Région2 et inversement, et si cet échange fait diminuer la frontière mutuelle. Cette étape va également lisser les frontières en plus de trouver le score minimal.

3.5 Résultats

Cette partie présente les résultats de la méthode approchée pour résoudre le problème du partitionnement.

3.5.1 Amélioration d'une solution initiale

Sur la figure 3.4a, il y a la représentation du partitionnement initial réalisé en se basant sur les centroïdes. Le partage est inégal, la plus petite région est composée de 374 cellules et la plus grande, en possède 559. Après une unique itération et un passage à l'algorithme swap, la plus petite région a 462 cellules et la plus grande en a 553. Le résultat est visible sur la figure 3.4b. Ce test a été réalisé avec la matrice des distances.

3.5.2 Comparaison : distance euclidienne ou matrice des distances

Les tableaux 3.1 et 3.2 indiquent respectivement les résultats des simulations avec la distance et avec la matrice des distances. Dans ces tableaux, il est indiqué :

1. le nombre de régions demandées
2. le temps nécessaire pour générer les résultats
3. la moyenne du nombre de cellules qu'il faudrait par région
4. la taille de la plus petite région générée et celle de la plus grande
5. les bornes idéales dans lesquelles devrait se trouver chaque région. Dans le cas d'un calcul avec une matrice des distances, la borne est de ±1% de la moyenne. Dans le cas des distances

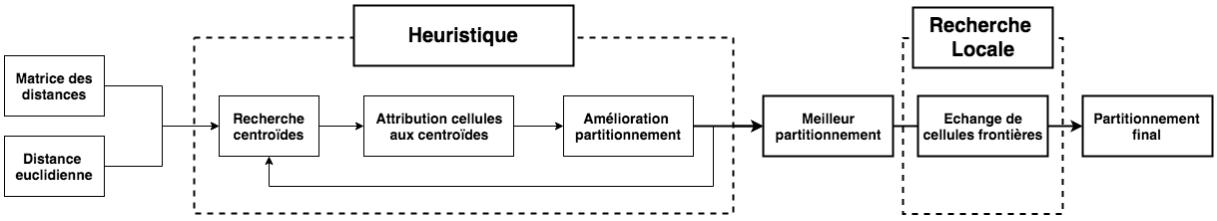
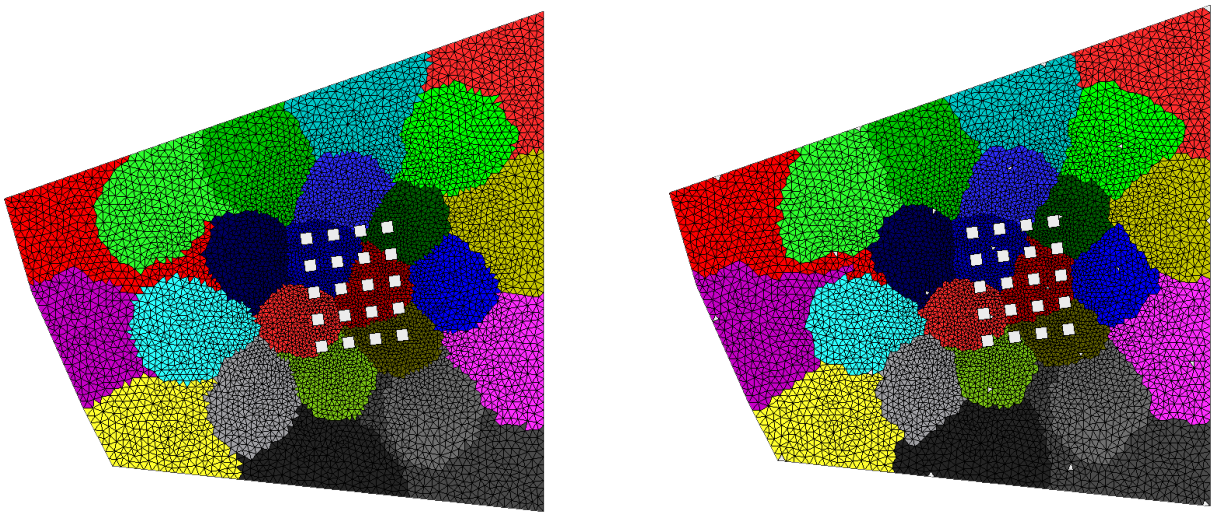


FIGURE 3.3 – Schéma général du partitionnement



(a) Partitionnement initial

(b) Amélioration du partitionnement

FIGURE 3.4 – Résultat de l'amélioration du partitionnement initial

euclidiennes, la borne est plus laxiste : ±3%. Ces bornes ont été définies expérimentalement sur base du cas test Tous.

6. la variance qui est calculée avec la formule : $\frac{\sum_{i \in 1..nRegion} (taille_i - \overline{taille})^2}{nRegion-1}$

Pour la table 3.2, la génération de la matrice des distances n'a pas été prise en compte, elle a été lue depuis un fichier de sauvegarde. Par contre, le temps de lecture a été pris en compte (10,6 secondes pour le fichier de Toce). Si la matrice des distances pour une certaine simulation n'a pas encore été sauvegardée, il faut rajouter ce temps à la durée totale. Pour la simulation Toce, le temps nécessaire aux calculs de cette matrice est de 81,5 secondes auxquelles il faut ajouter 9,9 secondes pour son enregistrement.

nombre de régions	temps total	moyenne	minimum	maximum	bornes	écart-type
4	4,6 s	2933	2932	2935	[2845 ;3021]	1,41
8	4,5 s	1466	1461	1469	[1422 ;1510]	2,57
24	13,2 s	489	464	548	[474 ;504]	16,78
48	33,1 s	244	120	334	[236 ;251]	38,33
96	98,71 s	122	50	223	[118 ;126]	24,65

TABLE 3.1 – Taille des régions générées à l'aide de la distance euclidienne pour Toce

nombre de régions	temps total	moyenne	minimum	maximum	bornes	écart-type
4	14,4 s	2933	2902	2993	[2903 ;2962]	42,43
8	18,3 s	1466	1451	1487	[1451 ;1480]	14,57
24	45,3 s	489	469	523	[484 ;493]	11,28
48	130,8 s	244	240	271	[241 ;246]	7,86
96	415,4 s	122	119	146	[120 ;123]	5,63

TABLE 3.2 – Taille des régions générées à l'aide de la matrice des distances pour Toce

La distinction entre un domaine généré avec la distance euclidienne et celui généré avec la matrice des distances est très rapide : la position des centroïdes dans le premier n'est pas du tout optimale. En effet, la carte de Toce divisé en 24 régions avec la distance euclidienne sur la figure 3.5b présente une forte concentration de cellules au centre de celle-ci, alors que l'extérieur n'est composé que du plus faible nombre de relativement grandes cellules. Les centroïdes sont représentés par des cellules blanches sur l'image. Sur la figure 3.5, tous les centroïdes sont présents sur la partie extérieure de la carte en laissant libre l'intérieur, pourtant, c'est à cet endroit qui est présent le plus de cellules. Lors de la répartition initiale, toutes les régions ont dû aller chercher les cellules libres au centre de la carte, en créant des couloirs de l'extérieur vers le centre.

La génération des centroïdes des figures 3.2 a tenu compte de la distance en termes du nombre de voisins séparant deux cellules. La répartition a été beaucoup plus homogène : il y a de nombreux centroïdes au centre de la carte et ceux sur l'extérieur sont plus distants les uns des autres.

Le premier point auquel il fallait veiller était le nombre de cellules par région. La méthode avec les distances euclidiennes offre de bons résultats avec un nombre de régions pas trop élevé. La table 3.2 prouve qu'en se basant sur répartition réelle des cellules sur le terrain, le nombre de cellules par région est très proche des bornes attendues.

Le second point important à respecter par notre programme de preprocessing était de trouver la frontière minimale. Ce dernier n'est pas capable de trouver la frontière minimale, par contre il essaye de la diminuer au maximum principalement grâce aux swaps qui lissent cette dernière. En se basant sur les résultats indiqués aux tables 3.3 et 3.4, la frontière est légèrement meilleure en employant la distance en terme de voisin. En se basant sur les figures 3.5 et 3.6, les régions ont une forme plus arrondie avec la matrice des distances. Pour trouver la surface avec la plus grande superficie et le plus petit périmètre, il faut employer un cercle.

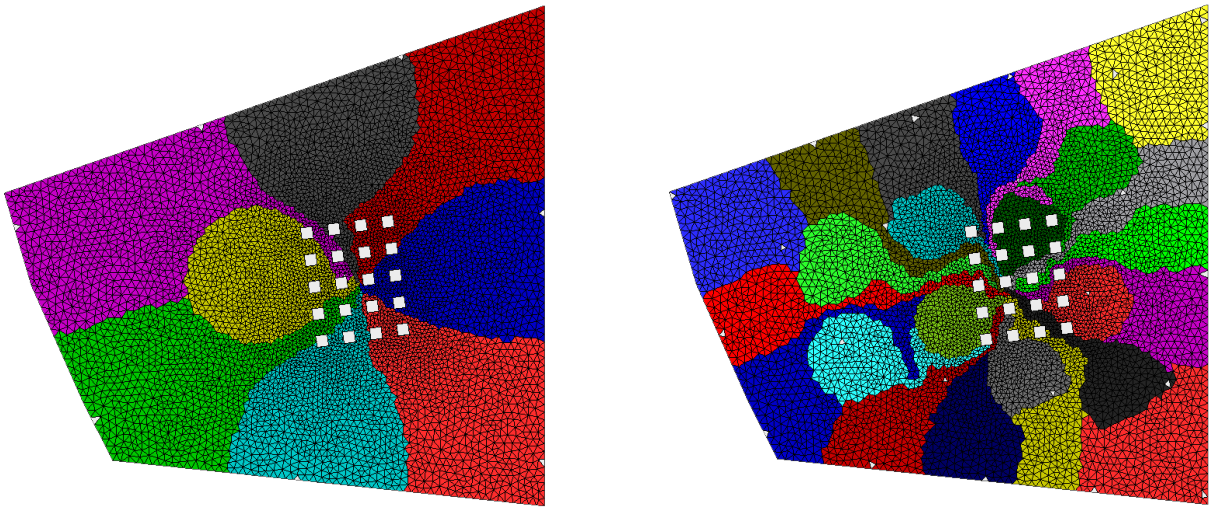
	Frontières		
nombre de régions	taille	taille avant swap	durée du swap
4	187	232	0,05 s
8	383	456	0,18 s
24	985	1146	3,02 s
48	1336	1593	19 s
96	1540	1825	59 s

TABLE 3.3 – Evolution de la taille des frontières avec la distance euclienne pour Toce

	Frontières		
nombre de régions	taille	taille avant swap	durée du swap
4	194	239	0,05 s
8	320	377	0,21 s
24	675	808	3,63 s
48	963	1168	14,65 s
96	1391	1635	56,42 s

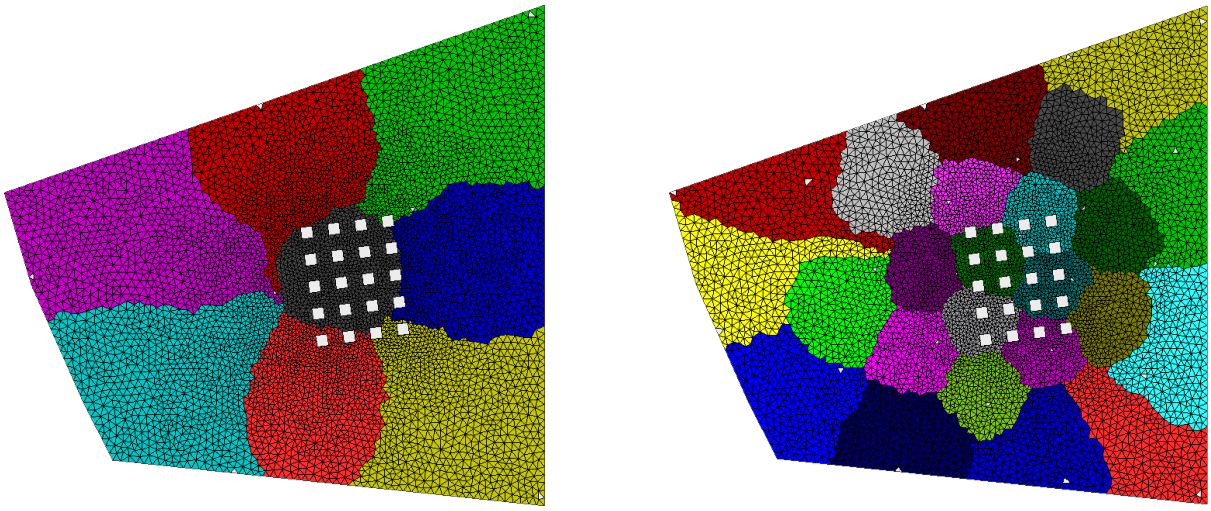
TABLE 3.4 – Évolution de la taille des frontières à l'aide de la matrice des distances pour Toce

Le dernier point à vérifier est le temps de calcul. De nombreuses tentatives de calculs ont été testées et celle qui fournissait le meilleur rapport temps de calcul contre résultat a été



(a) Toce en 8 régions (b) Toce en 24 régions

FIGURE 3.5 – Partition de Toce à l'aide de la distance euclidienne



(a) Toce en 8 régions (b) Toce en 24 régions

FIGURE 3.6 – Partition de Toce à l'aide de la matrice des distances

sélectionnée. Le temps de calcul en employant la distance euclidienne est bien meilleure que la seconde technique d'autant plus qu'elle ne doit pas calculer la matrice des distances.
Lorsque le nombre de régions est élevé, la phase de swap demande une grande partie du temps de calcul total. Pour découper le Toce en 96 régions en employant les distances euclidiennes, il faut 59 secondes sur un total de 98 secondes, ce qui représente 60% du temps total. Mais cette étape est importante pour améliorer le résultat obtenu.

3.5.3 Test sur un grand domaine

Tous [3] a été employé pour vérifier le bon fonctionnement du programme de séparation des cellules. La répartition de cette simulation sur son domaine est très inégale. En effet, il y a une zone, à une extrémité du domaine, qui correspond à un village, et contient de nombreuses cellules concentrées. À la seconde extrémité, les cellules représentent des surfaces beaucoup plus grandes. La figure 3.7 met en évidence cette dualité. La seconde difficulté liée à ce domaine est sa taille. En effet, il n'est pas possible de calculer la matrice des distances qui relie toutes les cellules entre elles. Il faut donc obligatoirement utiliser la distance euclidienne.

La table des résultats 3.5 indique des temps de simulation corrects pour un tel nombre de cellules. Par contre, le maximum et le minimum de tailles des régions sont fort éloignés des bornes attendues. Un point positif est que le maximum diminue presque d'un facteur 2 lorsqu'on double le nombre de régions. Ce nombre maximum est très important, car il correspond à la taille d'une région qu'un nœud du cluster va recevoir. Si ce nombre est très élevé, il mettra beaucoup plus de temps à calculer ses cellules correspondantes et ralentira tout le calcul global.

La différence entre les tailles des régions peut s'expliquer en analysant la carte de la simulation présente à la figure 3.7 : beaucoup de centroïdes vont se retrouver dans la partie basse gauche de la carte et n'auront que peu de cellules disponibles. De l'autre côté, près de la ville, il y aura énormément de cellules disponibles pour peu de centroïdes. Les régions de gauche vont donc essayer de s'étendre vers la droite pour prendre le surplus de cellules qui s'y trouve.

3.6 Tests unitaires

Des tests unitaires ont été ajoutés afin de s'assurer que les différents algorithmes utilisés dans le partitionnement soient corrects. Cette section indique les tests réalisés sur les fonctions qui

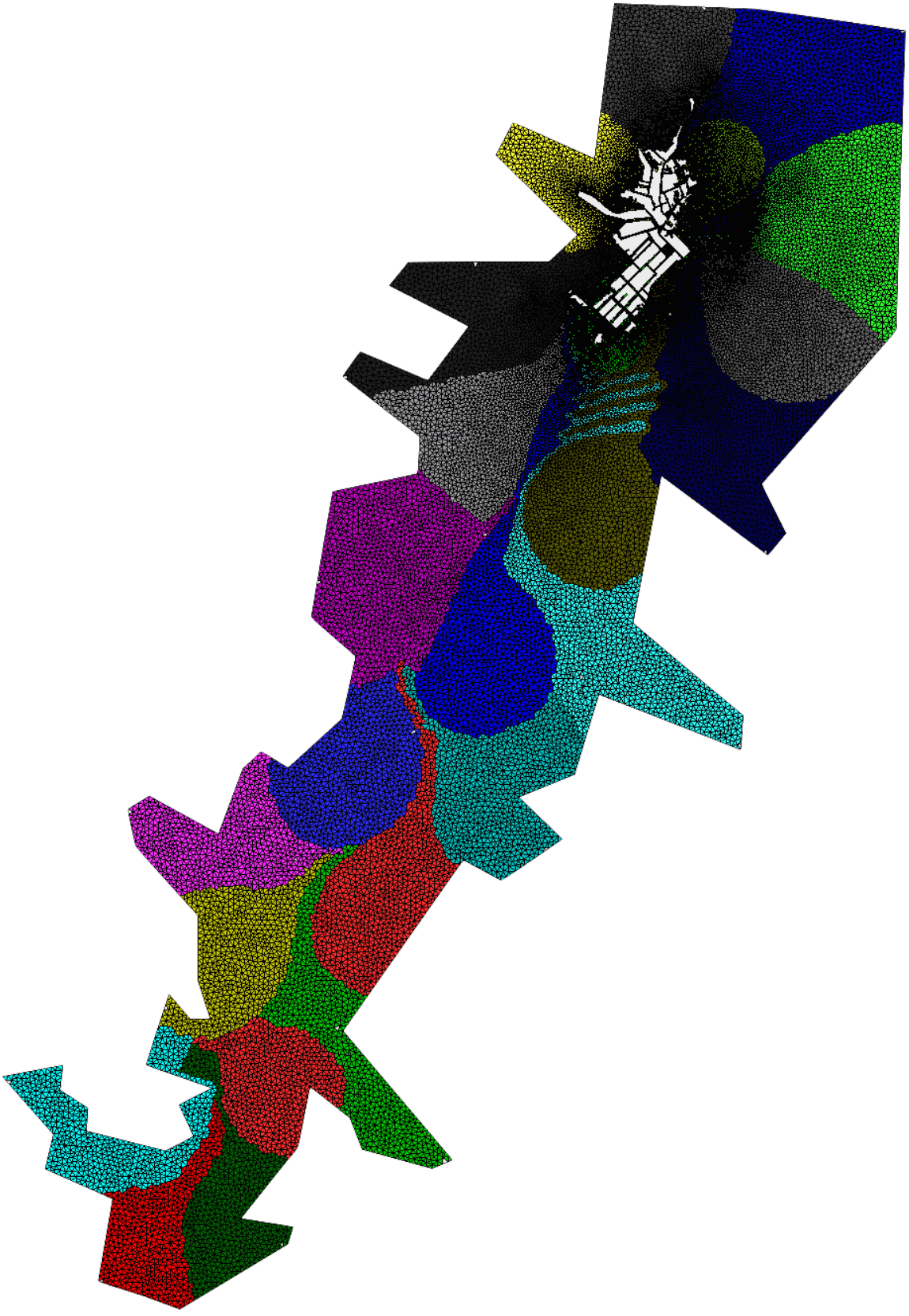


FIGURE 3.7 – Partitionnement du maillage de Tous en 24 régions

nombre de régions	temps total	moyenne	minimum	maximum	bornes	écart-type
4	84,3 s	15.227	13.377	15.849	[14.770 ;15.684]	1.224
8	77,3 s	7613	2756	8326	[7385 ;7842]	1.897
24	199,3 s	2537	875	4023	[2461 ;2614]	1.342
48	207,9 s	1268	268	2124	[1230 ;1307]	655
96	612,4 s	634	132	1196	[615 ;653]	346

TABLE 3.5 – Taille des régions du partitionnement de Tous

	Frontières		
nombre de régions	Taille avant le swap	Taille après le swap	Durée du swap
4	806	707	0,86 s
8	1574	1388	3,79 s
24	2468	2116	16,01 s
48	3624	3146	76,87 s
96	5195	4513	354,8 s

TABLE 3.6 – Évolution de la taille des frontières pour Tous

représentent un enjeu important pour le programme et s'assure en testant différents cas du bon fonctionnement de celles-ci dans toutes situations.

3.6.1 Chemin entre deux cellules

Sur le domaine représenté sur l'image 3.8a, les cas scénarios suivants sont testés :

— La cellule 1 va changer de région et les cellules 0 et 6 sont les voisins, sortie attendue : faux (car le domaine sera coupé en deux, avec la cellule 0 toute seule d'un coté et le reste du domaine bleu de l'autre)

— La cellule 8 va changer de région et les cellules 7 et 9 sont les voisins, sortie attendue : faux

— La cellule 14 va changer de région et les cellules 13 et 15 sont les voisins, sortie attendue : faux

— La cellule 26 va changer de région les cellules 21 et 27 sont les voisins, sortie attendue : faux

— La cellule 32 va changer de région les cellules 27 et 33 sont les voisins, sortie attendue : vrai

3.6.2 Déplacement des cellules entre régions

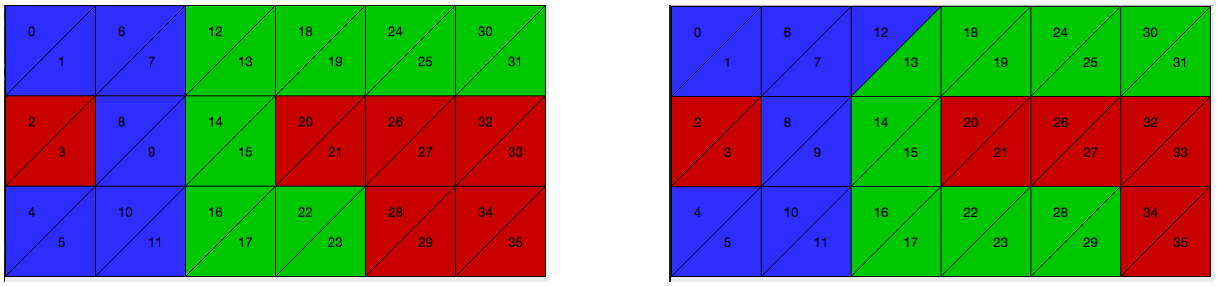
Ce test vérifie le bon fonctionnement de la fonction `moveCell`. Cette fonction prend en argument le numéro d'une cellule à déplacer, la région qui va perdre la cellule et celle qui va la recevoir, la liste des appartenances de toutes les cellules ainsi que la liste des frontières des deux régions. La fonction va déplacer la cellule en question dans son nouveau domaine et va modifier les frontières des deux régions. En se basant sur les deux images présentées sur la figure 3.8, la partie gauche représente le domaine avant les tests et la partie droite, le résultat.
Différentes actions sont testées :

— La région verte prend la cellule 26 à la région rouge

— La région bleue prend la cellule 12 de la région verte

— La région verte prend la cellule 29 à la région rouge

Entre chaque changement de région pour une cellule, la totalité des frontières de chaque région est comparée pour vérifier que les cellules qui ne sont plus frontière ont été enlevées et les nouvelles ont été ajoutées.



(a) Avant le déplacement des cellules

(b) Après le déplacement des cellules

FIGURE 3.8 – Domaine pour les tests des chemins

3.7 Conclusion

Le solveur Gurobi permet de trouver la meilleure répartition des cellules en régions, mais les calculs sont chronophages. Tandis que la méthode approchée permet d'obtenir des résultats satisfaisants. Les calculs sont relativement rapides même pour un domaine volumineux.

Si le domaine est limité et la mémoire RAM de l'ordinateur ne l'est pas, il est intéressant d'utiliser la matrice des distances qui est créée en comptant le nombre de voisins entre deux cellules. Grâce à cette matrice, les régions sont plus équilibrées surtout lorsqu'il s'agit d'une division en de nombreuses régions.

Si le domaine est volumineux et la mémoire RAM limitée, la distance euclidienne offre de bons résultats pour des divisions en un nombre de régions limité. Si le nombre de régions augmente, la valeur minimale et maximale du nombre de cellules dans les régions s'éloigne des bornes attendues. Malgré cet éloignement, si le nombre de régions attendues est doublé, la borne maximale diminue avec un facteur proche de 2.

4 Multithreading

Le multithreading est la capacité d'exécuter plusieurs tâches en parallèle. Dans un processus multithreadé sur un unique processeur, le processeur peut basculer entre les ressources des tâches, et crée artificiellement de la concurrence sur l'exécution. Pour le même processus multithreadé dans un environnement multiprocesseur, chaque tâche est exécutée sur un processeur différent (si le nombre de threads est inférieur ou égal au nombre de cœurs) ce qui entraîne une exécution parallèle. Au niveau de la programmation, cela se résume à utiliser deux types de librairies de multithreading que sont `libgomp` de GCC (implémentant le standard de programmation parallèle OpenMP) et POSIX Threads.

Ce chapitre est partagé en plusieurs sections. Dans la première section nous introduisons les notions et concepts entourant le multithreading. Ensuite, la deuxième section présentera le fonctionnement d'OpenMP et de POSIX Threads. La troisième partie de ce chpaitre aborde brièvement les optimisations du compilateur. Le quatrième point reprend les améliorations implémentées au sein du programme initial, suivi d'une dernière section analysant les performances des solutions développées.

4.1 Concepts de parallélisation

Un thread est une procédure (ou tâche) qui tourne indépendamment du programme principal [24]. Chaque thread possède ses propres variables privées et un accès à la mémoire partagée. Deux threads du même processus possédant un même pointeur vers une adresse mémoire vont pouvoir accéder à la même donnée. Par contre, il faut protéger son écriture. Si deux threads tentent de modifier la même donnée au même moment, le résultat sera aléatoire. En lecture, deux threads peuvent accéder au même moment à la même donnée sans créer de problème.

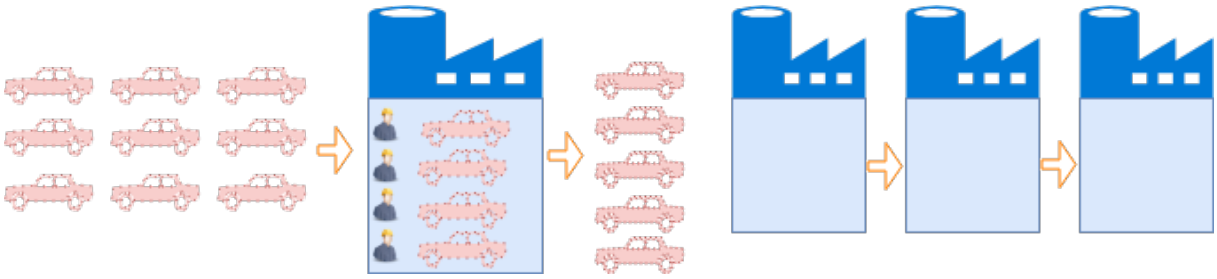


FIGURE 4.1 – Décomposition en domaine

Partitionnement du travail

Il existe deux grandes catégories de partage du travail : la décomposition en domaine et la décomposition fonctionnelle [25]. Pour visualiser cette décomposition, considérons un exemple de

la fabrication de 28 voitures. Les voitures représentent les données, les travailleurs représentent les threads, et les tâches à exécuter sont les différentes étapes de la construction de la voiture.

— La **décomposition en domaine** divise, pour une tâche précise, l'ensemble des données à traiter. Sur la figure 4.1, la décomposition en domaine consiste à appliquer la première tâche à toutes les voitures, et lorsque la première étape de fabrication est terminée pour toutes les voitures, la seconde tâche commence. Dans cette décomposition, le partage peut être soit en bloc (figure 4.2a) soit cyclique (figure 4.2b).

- Le **partage en bloc** divise le domaine en parts égales et chaque thread est responsable d'une partie. En revenant à notre exemple et en considérant 4 ouvriers, le premier ouvrier est responsable des 7 premières voitures, le second est responsable des voitures numéro 8 jusqu'à 15 et ainsi de suite.
- Le **partage cyclique** distribue à tour de rôle une tâche à chaque thread. Pour la production de voiture, l'ouvrier 1 est responsable des voitures [1,5,9,13...], l'ouvrier 2 doit produire les voitures [2,6,10,14...] et ainsi de suite.

(a) Partage en bloc (b) Partage cyclique

FIGURE 4.2 – Différence entre partage en bloc et partage cyclique

— La **décomposition fonctionnelle** (figure 4.3) assigne une tâche à un thread. Celui-ci traitera seul l'ensemble des données. Sur l'exemple des voitures, la décomposition fonctionnelle, la production serait à la chaîne : chaque ouvrier est responsable d'une tâche et traite une voiture à la fois. Dès que le premier ouvrier a fini sa tâche, le second ouvrier effectue la seconde tâche dessus (pendant ce temps-là, le premier ouvrier s'occupe d'une nouvelle voiture).

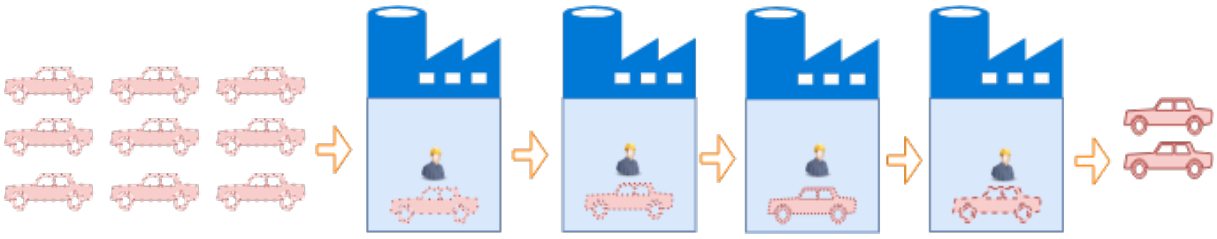


FIGURE 4.3 – Décomposition fonctionnelle

L'équilibrage de charge

Le Load Balancing est un terme anglais qui signifie la répartition de la charge du travail. Lorsqu'un certain travail est divisé en plusieurs threads, le temps pour effectuer cette tâche sera le temps que mettra le thread le plus lent à terminer la sienne. Pour cette raison, il est primordial de répartir correctement le travail pour que tous les threads finissent de travailler au même moment. La figure 4.4 montre une situation dans laquelle les tâches ne sont pas distribuées correctement.

Il existe deux manières de répartir le travail :

— La **répartition statique** divise au début les différentes tâches à effectuer par chaque thread. Cette division doit être la plus équitable possible. Parfois, il est impossible de prévoir une répartition équitable ; la répartition dynamique permet de gérer cette situation.

— La **répartition dynamique** répartit au fur et à mesure les tâches à effectuer entre les différents threads.

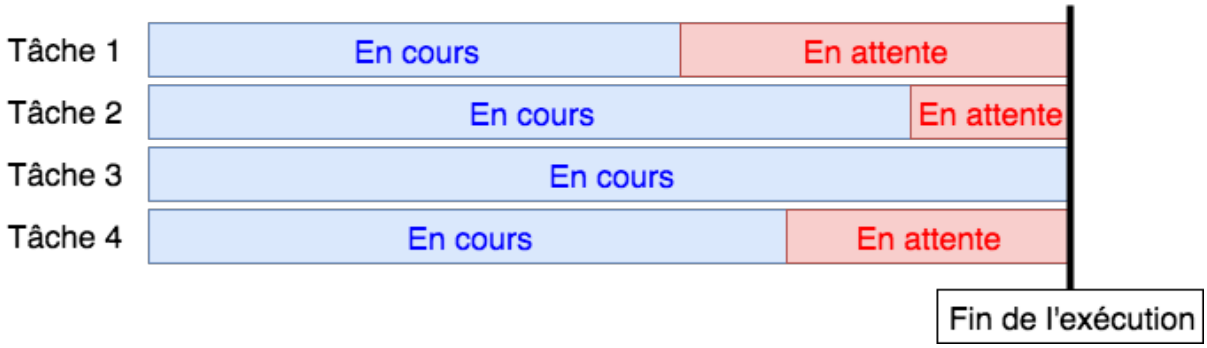


FIGURE 4.4 – Mauvaise distribution des tâches

Dans le cas ou la charge de travail n'est pas connue en avance pour équilibrer la charge de travail sur les différents threads, il est possible d'adapter en cours d'exécution la charge de travail de chaque thread. Sur la figure 4.4, le thread 1 qui effectue la tâche 1 termine en premier. Au lieu de rester en attente, il va voler une partie de la charge du travail d'un autre thread (tâche 3) pour alléger son exécution. Le thread 1 met en pratique le concept de work stealing terme anglais désignant le vol de la charge de travail.

Thread pool

Le **thread pool** permet de gérer le load balancing de manière dynamique et donc de distribuer les tâches entre différents threads de manière dynamique. Le processus principal est responsable de la distribution des tâches de telle sorte que lorsqu'un thread est disponible, il va chercher une nouvelle tâche. Toutes sortes de tâches peuvent être insérées dans la liste des tâches. Cette liste peut également être augmentée dynamiquement.

L'utilisation d'un thread pool permet de limiter le nombre de créations de threads. En effet, ce sont toujours les mêmes threads qui sont employés pour exécuter les différentes tâches.

Granularité

Un concept important de la parallélisation est la granularité du programme. Ce dernier est un ratio entre le temps passé à la communication et le temps de calcul. [25]

Si le programme est composé de beaucoup de petites tâches, et que ces tâches sont attribuées à beaucoup de threads, il est de type grain fin. Ce type de parallélisme est idéal pour maîtriser le load balancing. Par contre, si la granularité est trop fine, l'effet inverse se produit pour un thread pool : le programme va passer plus de temps à gérer la distribution des sous-tâches que le calcul des sous-tâches en elles-mêmes.

À l'opposé, le parallélisme à gros grain correspond à un programme avec peu de lourdes tâches calculatoires et quelques threads. Il y a peu de communications, mais malgré cela, ces programmes sont difficilement parallélisables.

4.2 Outils de parallélisation

Les deux libraires les plus communément utilisées dans le multithreading sont OpenMP et la librairie POSIX Threads. La première partie de ce point présente OpenMP, son fonctionnement ainsi que les paramètres utilisés dans les directives introduites dans le programme SFlow2D. La deuxième partie est consacrée à une brève explication du fonctionnement du multithreading par la libraire POSIX Threads.

4.2.1 OpenMP

OpenMP [26] est une spécification pour un ensemble de directives au compilateur, de routines libraires et des variables d'environnement. Cette spécification, implémentée au travers de la librairie `libgomp`, est utilisée pour spécifier du parallélisme haut-niveau. Plusieurs raisons justifient son utilisation :

— OpenMP est le standard le plus répandu pour les systèmes multiprocesseurs, supporte 3 langages de programmation (C, C++ et Fortran) et est implémenté par de nombreux fournisseurs de matériel informatique.

— OpenMP est relativement simple d'utilisation

— Beaucoup de recherches sont effectuées sur OpenMP, la gardant ainsi à jour avec les développements matériels les plus récents.

Fonctionnement

OpenMP fonctionne grâce à l'introduction de directives indiquant au compilateur comment les opérations doivent être exécutées. Le compilateur génère ensuite du code multithreadé et s'occupe d'assigner à chaque thread le code qu'il doit exécuter et les variables auxquelles il a accès ou pour lesquelles il faut créer des variables locales. De plus, OpenMP se charge de créer, suspendre, réveiller et mettre fin aux threads. Le compilateur ne vérifie pas si les directives vont effectuer le bon travail, c'est au développeur de s'assurer que les synchronisations soient insérées au bon endroit. Le résultat de la compilation est un programme multithreadé.

Paramètres

Toute directive OpenMP commence par `#pragma omp` suivi de paramètres se terminant par une nouvelle ligne. La directive avec le paramètre `parallel` démarre un bloc parallèle. Elle crée une équipe de N threads (où N est déterminé au lancement du programme), qui exécute les opérations dans la construction.

OpenMP est particulièrement utile dans l'exécution de boucles `for`. La directive `#pragma omp parallel for` crée une équipe de N threads. Chaque thread exécute une partie de la boucle. Le concept de load balancing, est également présent dans OpenMP sous 3 arguments possibles au paramètre schedule() :

schedule(static) assigne statiquement les itérations aux threads lorsque le programme entre dans une zone parallèle. Un scheduling statique risque de poser des problèmes dans le cas d'itérations de temps d'exécutions très hétérogènes.

schedule(dynamic) assigne dynamiquement les itérations aux threads, ce qui l'avantage dans les itérations qui prennent des temps différents. Malheureusement, cela génère des pertes de temps, car le thread s'arrête et attend une nouvelle valeur de la variable de boucle pour sa nouvelle itération.

schedule(guided) offre un compromis entre la répartition statique et dynamique. Il commence par de gros morceaux et s'ajuste ensuite à de plus petits morceaux si la charge de travail est déséquilibrée.

L'ordre dans lequel la boucle est exécutée n'est pas spécifié et dépend des conditions d'exécutions.

L'utilisateur peut également spécifier que certaines variables soient propres ou privées à un thread en ajoutant le paramètre `private(nomDeLaVariable)`. Si tout au contraire la variable doit être partagée avec les autres threads alors l'utilisateur introduit le paramètre `shared(variablePartagée)`. Lorsque la variable est partagée il faut s'assurer qu'un seul thread mette à jour la variable, cela se fait soit par une nouvelle directive `#pragma omp critical` ou par `#pragma omp atomic`. L'option `critical` s'assure qu'un seul thread n'exécute que cette **partie** de code. L'option `atomic` sur une opération quant à elle, lui s'assure qu'aucun autre thread n'intervienne lors de la mise à jour de la valeur.

4.2.2 Pthread

La libraire POSIX Threads ou souvent appelée `pthread`, est une implémentation de l'interface de programmation normalisée en langage C spécifiée par la norme IEEE POSIX 1003.1c. Pthread permet de faire de la parallélisation bas-niveau : elle permet de gérer les threads et la synchronisation des données en mémoire. L'exemple ci-dessous [27] permet de visualiser les concepts introduits précédemment ainsi que les notions de création de threads, terminaison de threads et verrou d'une ressource.

```
int sum;
pthread_mutex_t mutex_sum;
void *maFonctionThread(void *args)
{
    int err;
    long tid = (long) args;
    printf("Hello World de thread %d",tid);
    err=pthread_mutex_lock(&mutex_global);
    sum = sum + tid;
    err=pthread_mutex_unlock(&mutex_global);

   ...
}
int main (int argc, char *argv[])
{
    int tid, err;
    sum = 0;
    pthread_t monThread[NUM_THREADS];
    for(tid = 0; tid < NUM_THREADS; tid++)
        pthread_create( &monThread[tid], NULL, maFonctionThread, (void *) tid);
    for(tid = 0; tid < NUM_THREADS; tid++)
       err = pthread_join(monThread[tid], NULL);
    ...
}
```

Dans ce morceau de code, plusieurs threads sont lancés à l'aide de la commande `pthread_create` et exécutent la fonction `maFonctionThread` chacun avec leur propre argument tid : leur numéro de thread. Afin d'obtenir la somme de tous les numéros de threads, le thread va mettre à jour la variable `sum`. Il est important que la mise à jour de la ressource ne se fasse pas en même temps. Le thread acquiert un verrou, implémenté au travers de la variable `mutex_sum`, sur la ressource `sum` en faisant appel à la fonction `pthread_mutex_lock`. S'il n'y a pas accès, il doit attendre et essaie à nouveau plus tard, autrement il la met à jour et libère le verrou en appelant

`pthread_mutex_unlock`. Par la suite, la fonction `pthread_join` est appelée sur chacun des threads ce qui fait que l'exécution principale ne continue pas tant que les threads n'ont pas fini. Les threads sont ainsi arrêtés et toutes les tâches sont finies.

Un dernier concept important concerne la variable de condition [28]. C'est un mécanisme de synchronisation qui permet à un thread de suspendre son exécution jusqu'à ce que la condition soit vérifiée. La première étape consiste à initialiser cette variable de condition à l'aide de `pthread_cond_init`.
Ensuite un thread va se mettre dans un état de pause en attendant la vérification de la condition en employant `pthread_cond_wait`. Cette fonction déverrouille également un mutex qui a dû être précédemment verrouillé par ce même thread. Lors de cette attente, le processus ne consomme pas de ressources de calcul. Plusieurs threads peuvent se retrouver bloqués par une même condition.

La fonction `pthread_cond_broadcast` permet d'envoyer, à tous les threads en attente d'une certaine condition, un signal qui va les libérer. Si aucun thread n'était bloqué par cette variable de condition, il ne se passe rien.
`pthread_cond_signal` est similaire à `pthread_cond_broadcast`, mais il ne libère qu'un seul thread. Si plusieurs threads étaient en attente, seul un d'eux est aléatoirement libéré.

4.3 Optimisations du code

Avant que le programme parallélisé ne s'exécute, son code source doit être traduit par le compilateur en un langage machine et compilé sous forme d'un exécutable. L'objectif du compilateur lorsque le programme est compilé sans l'option d'optimisation (-O0) est de réduire le coût de compilation et de faire en sorte que le débogage produise les résultats escomptés [29]. En effet, lorsque le développeur débogue le code en arrêtant l'exécution par un point d'arrêt (breakpoint) entre des opérations, il peut modifier et assigner de nouvelles valeurs aux variables et obtenir un résultat tel attendu par l'exécution du code.

Si au contraire, le programme est compilé avec l'option d'optimisation `-O` suivi d'un chiffre indiquant son niveau d'optimisation, le compilateur tentera d'améliorer la performance et la taille du code au détriment du temps de compilation et de la possibilité de déboguer le programme. Par conséquent, le compilateur effectue des optimisations grâce aux informations obtenues lors de la compilation de multiples fichiers en un exécutable, contrairement à la compilation fichiers par fichiers où les liens entre les informations des fichiers ne sont pas créés.

Il existe trois niveaux d'optimisation : -O1, -O2 et -O3. La compilation des différentes versions améliorées deSFlow2D se fait avec -O3 produisant ainsi de meilleurs résultats que les autres optimisations. Trouver le meilleur niveau d'optimisation est propre à chaque programme, pour certains, l'usage de -O2 fournit de meilleurs résultats.

4.3.1 Vectorisation

La vectorisation [30] transforme un programme appliquant une opération sur une seule donnée à la fois en un programme capable de traiter cette même opération sur plusieurs données en même temps. Les CPUs modernes offrent la possibilité d'effectuer des opérations vectorielles lorsqu'une instruction est appliquée à plusieurs données. Cette parallélisation porte le nom de Single Instruction Multiple Data (SIMD). L'avantage de ce processus est donc de pouvoir traiter plus de données, jusqu'à 16 simultanément, avec une même opération. Ce nombre varie en fonction du modèle et de la gamme du processeur.

Cela requiert toutefois de transformer le code dont la structure est un ensemble de vecteurs d'objets en un ensemble d'objets de vecteurs. OpenMP bénéficie également de cette optimisation avec l'option `simd`.

Dans les résultats, présentés à la section 4.5, nous ferons donc la part entre les programmes compilés avec -O3 et sans -O3.

4.4 Parallélisation

Dans ce point, nous présentons tout d'abord les différentes étapes d'exécution parallélisées de SFlow2D à partir des outils et des concepts introduits dans les points 4.1 et 4.2. La figure 4.5 sert de guide pour la suite des explications.

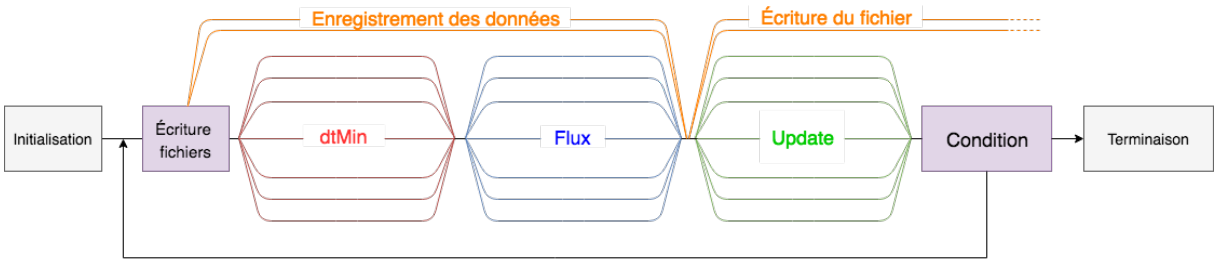


FIGURE 4.5 – Le cycle des threads

Le code parallélisé suit une décomposition en domaine. En effet, tous les threads calculent d'abord le nouveau pas de temps puis le flux et enfin la mise à jour des cellules. Cependant, l'écriture de fichiers de résultats suit la décomposition fonctionnelle. Un thread sera uniquement responsable de l'écriture d'un fichier. Analysons en premier la parallélisation du calcul du flux, la mise à jour des cellules, le calcul du nouveau pas de temps et ensuite l'écriture des fichiers.

4.4.1 Flux

Un point primordial de cette fonction est la protection des données partagées. Chaque cellule possède une variable, le buffer du flux, qui accumule la quantité d'eau qui rentre ou sort de la cellule au pas de temps actuel.

Lors du calcul du flux sur une interface, les buffers de flux de la cellule droite et de la cellule gauche seront modifiés. Dans certains cas, il est possible que deux threads calculent en même temps deux interfaces d'une même cellule. Ils risquent donc de modifier en même temps le buffer de flux. Pour cette raison, un verrou est mis sur cette variable. Juste avant la modification de la valeur, le thread prend le *lock* (verrou) de la cellule en question, modifie la valeur et ensuite libère le lock. La mise à jour des buffers de flux des deux cellules adjacentes se fait indépendamment pour éviter de se retrouver dans une situation de deadlock (situation dans laquelle plusieurs threads verrouillent au moins un lock et sont en attente pour d'autres locks verrouillés par d'autres threads. Ils se bloquent donc mutuellement).

Une version sans lock a également été développée. À l'initialisation des cellules, la variable du buffer de flux est remplacée par un tableau de buffer de flux. La taille du tableau est égale au nombre d'interfaces qu'elle possède. Chaque interface reçoit pour chaque cellule un numéro. Celui-ci correspond à la case dans laquelle il ira écrire son buffer de flux. Ce numéro est unique pour un même tableau, il n'est donc pas possible que deux threads tentent de modifier la même variable au même moment.

Pthread

Lors du calcul du flux, plus particulièrement pour le multithread avec pthread, un certain nombre de threads sont créés par un appel à `pthread_create` où l'argument correspond au numéro du thread. Grâce à son nombre unique, il saura quelle interface il doit calculer. Le partage des tâches se fait de manière cyclique : si une simulation possède X threads, le thread numéro N (nombre allant de 0 à X-1) devra calculer les interfaces $N+X*k$ (avec k, un nombre démarrant à 0).

Un partage spatial aurait pu être employé pour profiter de la localité d'accès à la mémoire, mais en raison de l'hétérogénéité de la topologie et de la numérotation des cellules ce choix semblait moins raisonnable. En effet, lorsqu'une cellule ne possède pas encore d'eau, les calculs sont fortement diminués. Il est impossible de savoir à priori la distribution des cellules sur la carte et il est fort probable que par mesure préventive, la région représentée soit plus grande que la région inondée. Donc une partie de ces cellules seront vides d'eau et donc très rapides à calculer au moins à un moment de la simulation. Avec le partage cyclique, les cellules sont réparties le plus équitablement possible entre les threads et il n'y a que peu de probabilité qu'un thread ne doive calculer que des cellules vides. Par contre, si les cellules sont bien ordonnées, l'avantage de la localité spatiale est diminué. Par conséquent, une approche cyclique a été adoptée en vue d'équilibrer la répartition de la charge de travail parmi les threads.

OpenMP

Le calcul du flux multithreadé à l'aide d'OpenMP, utilise un scheduling avec l'option `schedule(guided)`. Ainsi, le retard qui serait accumulé par des calculs plus longs sera rééquilibré par le scheduler. Ce type de partage offre les meilleures performances.

4.4.2 Update

Le calcul de la mise à jour des cellules se base sur le buffer de flux rempli à l'étape précédente. Pour distribuer les cellules entre les threads, le partage cyclique est employé pour les mêmes raisons citées dans le point 4.4.1.

Chaque cellule est indépendante, il n'y a donc pas de risques de modifier les mêmes variables au même moment.

— Si un unique buffer de flux est employé par cellule, il faut seulement mettre à jour les différentes variables de la cellule et réinitialiser la valeur du buffer.

— Si un tableau de flux buffer est employé, il faut d'abord additionner les différentes valeurs du tableau pour ensuite mettre à jour la cellule. La dernière étape consiste à réinitialiser toutes les valeurs du tableau.

Le tableau de buffer de la cellule permet d'éviter l'emploi des locks mais alourdit la mise à jour de cette dernière.

4.4.3 dtMin

Le calcul du nouveau pas de temps par la fonction `dtMin` est également parallélisable. Différents threads parcourent la liste des cellules. Chaque thread est responsable d'un morceau de la liste. Ils possèdent également des variables privées qui leur permettent d'enregistrer le minimum de leur morceau de liste. Lorsque tous les threads ont terminé de trouver leur minimum local, ils les comparent pour trouver le minimum global.

4.4.4 Écriture de fichiers

L'écriture des fichiers est une partie de l'exécution bloquante : tant que l'écriture de tous les fichiers n'est pas terminée, l'exécution des calculs de flux et les mises à jour de cellules sont mises en pause. Pourtant l'écriture de fichiers sur le disque dur peut être très lente. Des threads additionnels à ceux qui effectuent les calculs ont été ajoutés pour permettre de rendre cette partie parallèle.

À chaque pas de temps, le thread principal est toujours responsable de vérifier s'il n'est pas temps d'écrire un nouveau fichier. Si un des fichiers doit être généré, il lance un thread qui va s'en charger. Il lance autant de threads qu'il faut écrire de fichiers différents à ce moment-là.

La première chose que le nouveau thread va effectuer est l'enregistrement dans un tableau les données qui risquent d'être modifiées. Écrire en mémoire est beaucoup plus rapide que d'écrire sur le disque dur s'il reste suffisamment d'espace dans la mémoire RAM. Par exemple pour générer une instantanée, il faut indiquer pour chaque cellule les 3 coordonnées du centre et les 3 variables hydrauliques. Les coordonnées des nœuds restent inchangées d'une itération à l'autre, mais par contre les variables hydrauliques risquent de changer à chaque itération. Ce sont donc ces 3 dernières variables qui sont enregistrées.

Pendant que le thread secondaire enregistre ces variables, l'exécution principale est autorisée à calculer le nouveau pas de temps et le flux, car ces étapes n'impactent que le buffer du flux et non les variables hydrauliques. Après cela, le thread principal attend que tous les threads d'écriture aient fini l'enregistrement (c'est souvent très rapide). Dès que tous les threads d'écritures ont indiqué qu'ils ont fini d'enregistrer leurs données, le thread principal continue à s'exécuter normalement.

Dès que les threads d'écriture ont terminé d'enregistrer les données, ils écrivent les données sauvegardées et les données de la mémoire partagée, puisqu'inchangées, dans un fichier. Les threads d'écriture ne bloquent pas l'exécution du thread principal. La seule condition pour laquelle un thread d'écriture risque de bloquer le thread principal est s'il n'a pas fini d'enregistrer son fichier et qu'on lui demande d'en enregistrer un nouveau du même type. Ce cas extrêmement rare n'apparaît que si la fréquence de demande de génération de fichiers est très élevée. Si c'est le cas, le thread principal attend qu'il finisse d'écrire son premier fichier.

4.4.5 Thread pool

Dans le programme SFlow2D, nous avons implémenté un thread pool personnalisé. Le processus principal est responsable d'orchestrer les différents threads lors de l'exécution. À l'initialisation du programme, un nombre fixé de threads sont créés, ce seront les travailleurs.

La première tâche à effectuer est le calcul des flux aux interfaces. Le thread principal va donc initialiser le compteur au nombre d'interfaces et réveiller tous les travailleurs. Ceux-ci vont verrouiller le compteur et le lire, si celui-ci est supérieur à -1, ils vont diminuer puis débloquer le compteur et enfin traiter l'interface qui correspond à la valeur lue. Lorsque la pile d'exécution est vide (le compteur atteint -1), ils se bloquent avec un `pthread_cond_wait` en attente de nouvelles tâches. Le thread principal va attendre que chaque travailleur ait fini ses tâches avant de continuer la suite de l'exécution.

Le programme SFlow2D peut être considéré comme un problème de parallélisation fine, donc le traitement d'une interface à la fois n'est pas très efficace. Pour cette raison, un travailleur peut prendre plusieurs interfaces à traiter à la suite sans retourner au compteur. Il y a plusieurs paliers : 1 000, 100 et 1. Si le compteur vaut X et est supérieur à 1001, il va diminuer le compteur

de cette valeur et il va calculer les interfaces allant de $X - 1\ 000$ jusqu'à X. Si le compteur est compris entre 1 000 et 101, il va traiter 100 interfaces à la suite. Ou sinon, il les traite une par une.

La seconde étape est la mise à jour des cellules. Le thread principal va mettre à jour la valeur du compteur au nombre de cellules et il va changer la valeur d'un booléen. Ce dernier indique aux travailleurs s'ils doivent traiter le nombre qu'ils piochent comme le numéro d'une interface à calculer ou celui d'une cellule à mettre à jour. Le même principe de palier est employé (c'est normal vu que c'est le même thread pool).

Lorsque la simulation est terminée, le thread principal diminue le compteur à -2. S'il atteint une telle valeur, tous les threads auxiliaires savent qu'ils doivent se terminer.

Le système de compteur est, pour ce programme, beaucoup plus efficace à gérer qu'une liste de tâches à effectuer. Il suffit de mettre à jour le compteur et la variable booléenne. Une liste permet de gérer différentes sortes de tâches simultanément et d'en ajouter dynamiquement, dans ce programme ces fonctionnalités ne sont pas utiles.

4.5 Résultats et Analyse

Dans cette partie, nous testons les différentes améliorations proposées pour en comparer les performances. Sur base de ces résultats, nous sélectionnons les deux meilleurs configurations et les testons sur le cas test Tous.

4.5.1 Benchmark

Dans le tableau 4.1, le résumé de tous les temps d'exécution y sont repris. La première colonne correspond au nombre de threads et à la version employée :

- MT+update+flux+lock : Multithread sur l'update et le flux, les locks sont employés pour protéger l'écriture des buffers de flux
- MT+update+flux-lock : Multithread sur l'update et le flux, il n'y a pas de locks. Un tableau est présent pour gérer les buffers de flux
- TP+update+flux+lock : Thread pool employé sur l'update et le flux, des locks sont employés
- OpenMP+update+flux+lock : OpenMP sur l'update et le flux, les locks sont employés pour protéger l'écriture des buffers de flux
- OpenMP+update+flux-lock : OpenMP sur l'update et le flux, il n'y a pas de locks. Un tableau est présent pour gérer les buffers de flux
- OpenMP+lock+écriture : OpenMP sur l'update et le flux, il y a des locks. L'écriture des fichiers est multithreadée.

La deuxième et troisième colonne indique respectivement la moyenne du calcul du flux et celui de l'update. Ce nombre correspond à une moyenne sur 10 exécutions de la simulation test Toce. Le nombre entre parenthèses correspond à l'écart type. La dernière colonne correspond à la moyenne du temps pour l'exécution totale de la simulation. Cette dernière a été mesurée indépendamment des deux précédentes. En effet, pour obtenir des temps précis pour l'update et le flux, il a fallu modifier le code pour enregistrer le temps pris par ces deux actions à chaque itération. Ces modifications impactaient le temps total.
Les tests ont été effectués sur une machine de l'UCL qui possède 2 processeurs `Quad-Core AMD Opteron(tm) Processor 2356`, ce qui fait donc un total de 8 cœurs. Elle peut employer jusqu'à 32 Go de RAM.

Version de base

La version de base optimisée à l'aide de l'option `-O3` lors de la compilation affiche déjà des gains de performance significatifs. Les résultats présent dans le tableau sont comparés à la version de base sans optimisation.

4.5.2 Parallélisations supplémentaires

Deux nouvelles versions ont été testées avec des améliorations supplémentaires. Le code d'OpenMP+lock sur l'update et le flux ont été améliorés pour obtenir une nouvelle version : OpenMP avec un lock et le calcul parallélisé du dtMin.

Une seconde version avec OpenMP a été testée, elle est identique à la première sauf que la gestion parallèle des écritures des fichiers a été ajoutée.

L'ajout de la parallélisation du pas de temps minimal dans le code OpenMP avec les locks a permis d'obtenir un gain de 1,11. Lorsque l'écriture des fichiers est ajoutée, ce gain passe à 1,13 par rapport au précédent et 1,25 avec celui de base.
Le calcul du pas de temps minimal est une étape séquentielle au reste des itérations. Mais l'écriture des fichiers peut se faire en parallèle de la boucle principale.

Version	$\hat{\mu}_{flux}$ ($\hat{\sigma}_{flux}$)	$\hat{\mu}_{update}$ ($\hat{\sigma}_{update}$)	$\hat{\mu}_{tempstotal}$ (speed-up)
Base sans optimisation	**10,31 ms (0,44 ms)**	**8,34 ms (0,33 ms)**	**219,09 s (x 1,00)**
Base avec -O3	**9,42 ms (0,31 ms)**	**7,71 ms (0,23 ms)**	**135,93s (x 1,61)**
2 Threads			
MT+update+flux+lock	11,24 ms (0,14 ms)	5,38 ms (0,07 ms)	141,33 s (x 1,55)
MT+update+flux-lock	8,88 ms (0,06 ms)	6,93 ms (0,11 ms)	136,48 s (x 1,61)
TP+update+flux+lock	8,34 ms (0,19 ms)	4,58 ms (0,09 ms)	108,25 s (x 2,02)
OpenMP+update+flux+lock	7,85 ms (0,18 ms)	4,53 ms (0,06 ms)	102,43 s (x 2,14)
OpenMP+update+flux-lock	6,48 ms (0,13 ms)	5,97 ms (0,11 ms)	102,19 s (x 2,14)
OpenMP+lock+écriture	7,94 ms (0,16 ms)	4,58 ms (0,07 ms)	**96,38 s (x 2,27)**
4 Threads			
MT+update+flux+lock	7,34 ms (0,04 ms)	3,57 ms (0,02 ms)	97,40 s (x 2,25)
MT+update+flux-lock	6,0 ms (0,04 ms)	4,48 ms (0,05 ms)	94,35 s (x 2,32)
TP+update+flux+lock	5,29 ms (0,03 ms)	2,73 ms (0,03 ms)	71,78 s (x 3,05)
OpenMP+update+flux+lock	4,55 ms (0,14 ms)	2,65 ms (0,03 ms)	64,21 s (x 3,41)
OpenMP+update+flux-lock	3,92 ms (0,08 ms)	3,36 ms (0,05 ms)	64,30 s (x 3,41)
OpenMP+lock+écriture	4,51 ms (0,12 ms)	2,66 ms (0,04 ms)	**58,28 s (x 3,76)**
8 Threads			
MT+update+flux+lock	5,72 ms (0,08 ms)	2,83 ms (0,02 ms)	76,27 s (x 2,87)
MT+update+flux-lock	5,31 ms (0,01 ms)	3,71 ms (0,03 ms)	78,60 s (x 2,79)
TP+update+flux+lock	3,90 ms (0,03 ms)	2,12 ms (0,02 ms)	56,56 s (x 3,87)
OpenMP+update+flux+lock	3,01 ms (0,05 ms)	1,78 ms (0,04 ms)	45,63 s (x 4,80)
OpenMP+update+flux-lock	2,84 ms (0,10 ms)	2,22 ms (0,06 ms)	47,41 s (x 4,62)
OpenMP+lock+écriture	3,10 ms (0,06 ms)	1,84 ms (0,04 ms)	**42,26 s (x 5,18)**

TABLE 4.1 – Temps d'exécution décomposé des différentes versions améliorées et l'accélération par rapport au programme initial -O0

La version non parallélisée de l'écriture de fichiers oblige le programme à attendre que l'écriture soit finie avant de continuer la boucle principale bien que la boucle principale n'interagit plus avec les données de l'écriture. En conséquence, si le programme le plus performant est augmenté de ces deux améliorations (donc réduction du temps passé à calculer le `dtMin` et exécution en parallèle de l'écriture du fichier), il ne peut qu'en être plus performant.

Version	**Temps d'exécution**	**Speed-up**	**Efficacité**
8 Threads			
OpenMP+lock	45,63 s [0,94 s]	x 4,8	0,6
OpenMP+lock+dtMin	41,09 s [1,08 s]	x 5,33	0,667
OpenMP+lock+écriture+dtMin	**36,45 s** [0,84 s]	**x 6,01**	**0,751**

TABLE 4.2 – Temps d'exécution des versions les plus performantes de OpenMP avec les parallélisations supplémentaires sur Toce

Les résultats repris dans le tableau 4.2 confirment bien les affirmations ci-dessus. L'écriture parallélisée combinée à la parallélisation du calcul du pas de temps minimal fait que la version OpenMP+lock+écriture+dtMin se rapproche fortement de la limite théorique calculée au moyen de la loi d'Amdhal (voir 2.1.5).

La meilleure version d'OpenMP, avec le lock, le calcul du dtMin et la parallélisation des fichiers de sorties, a été testée sur le cas de test Tous. Il est comparé à un thread pool qui a subi les mêmes améliorations. Ces résultats sont présents à la figure 4.3.
Les deux dernières colonnes comparent le gain par rapport au code de base respectivement sans optimisation du compilateur et avec optimisation.

Version	**Temps d'exécution**	**Speed-up -O0 (Efficacité)**	**Speed-up -O3 (Efficacité)**
de base			
de base -O0	84h 8min 1s		
de base -O3	54h 38s	x 1,56	
8 Threads -O3			
OpenMP	**14h 41m 58s**	**x 5,72 (0,715)**	**x 3,67 (0,459)**
Thread pool	18h 23min 8s	x 4,58 (0,572)	x 2,94 (0,367)

TABLE 4.3 – Temps d'exécution des versions les plus performantes avec les parallélisations supplémentaires sur Tous

4.6 Notes sur l'état de l'art

L'article [4] suggérait d'utiliser l'option `schedule(dynamic)` avec une taille adaptée de la prise en charge des tâches par les threads. Toutefois, `schedule(guided)` offre de meilleurs résultats, car la redistribution des tâches est intrinsèque à la taille du problème.

5 Parallélisation sur cluster

Ce chapitre traite de la parallélisation du programme en employant un cluster. Un cluster est un ensemble d'ordinateurs ou serveurs connectés entre eux par un réseau extrêmement rapide pour former une entité capable de développer une grande puissance de calcul. Chaque unité de calcul représente un nœud. Pour profiter de cette puissance, il faut subdiviser les tâches en petites sous-tâches, réparties sur chaque nœud. Le protocole d'échange de messages *MPI* (Message Passing Interface) va être employé pour que les nœuds puissent communiquer les résultats intermédiaires entre eux [31, 32].

Chaque nœud possède ses propres variables, mais tous les nœuds emploient le même code. Ce concept est connu sous le nom de *SPMD* (Single Program Multiple Data). MPI permet l'utilisation d'une mémoire partagée entre différents processus, mais celle-ci est inefficace. Par la suite, nous n'allons pas considérer cette option, tous les nœuds auront donc leur propre mémoire.

5.1 Théorie

La première étape est l'initialisation de l'environnement en appelant `MPI_Init()`. De même, à la fin de l'exécution, il faudra mettre fin à l'environnement avec `MPI_Finalize()` pour désactiver ce dernier.

À l'initialisation, un communicateur est assigné par défaut, il s'agit de `MPI_COMM_WORLD`, il englobe tous les processus. Celui-ci sélectionne les nœuds qui recevront un message. Le communicateur fonctionne comme le canal d'une radio. La fonction `MPI_Comm_rank()` renvoie le rang de chaque processus compris entre 0 et $N-1$ (N étant le nombre de nœuds demandés). Ce numéro est unique à chaque nœud et les différencie.

Par exemple, dans le code, il pourra y avoir une ligne qui dit que si l'identifiant du nœud est égal à 0, il imprime un message. Ce message sera donc imprimé une seule fois par le nœud 0. Ces numéros servent également d'adresse pour envoyer les messages.

5.1.1 Les messages entre deux nœuds

Le transfert des messages est géré par MPI. Un message est constitué de paquets de données qui sont accompagnés d'un en-tête. Cette dernière donne des informations sur le message telles que : la source, le destinataire, la taille et le type de données envoyées.

Pour envoyer un message, il faut employer la fonction `MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)`. Dans cette fonction, il faut indiquer les paramètres suivants :

— `data` est un pointeur vers les données à envoyer.

— `count` est la taille des données à envoyer sous forme de tableau.

— datatype est le type des données envoyées. Il existe plusieurs types prédéfinis, tels que : MPI_INT ou MPI_DOUBLE, pour envoyer un nombre entier ou un nombre à virgule. Il est aussi possible de créer de nouveaux types plus complexes qui permettent d'envoyer différentes sortes de variables en même temps.

— `destination` est l'identifiant unique du destinataire au sein du communicateur.

— `tag` est une balise pour identifier un message.

— `communicator` est le communicateur dans lequel le message circulera.

Lorsqu'un nœud envoie un message, il faut que le destinataire appelle la fonction `MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)`. Les paramètres sont fort semblables à la fonction d'envoi. Pour les données, il faut préalablement allouer un espace assez grand dans la mémoire. Le compteur indique la taille maximale du message (si le message est plus grand qu'indiqué, la suite est ignorée). Les messages ne seront reçus que si l'expéditeur, le tag et le communicateur correspondent dans le message envoyé. Des mots clés permettent de remplacer certains de ces paramètres.
Il existe entre autres :

— `MPI_ANY_SOURCE` permet de recevoir un message de n'importe quel expéditeur

— `MPI_ANY_TAG` permet de ne pas tenir compte de la balise du message

Le `statut` du message peut s'avérer utile si un des mots-clés ci-dessus est employé. Le statut est une structure qui permet de récupérer la taille réellement envoyée, la balise et l'expéditeur.

Les deux fonctions `MPI_Send` et `MPI_Recv` sont bloquantes. Cela signifie que, lors de l'appel à une de ces fonctions, le processus sera bloqué tant que l'action opposée n'aura pas été appelée. Par exemple, si le nœud numéro 1 envoie un message au nœud numéro 2, le nœud 1 sera bloqué jusqu'à ce que le nœud 2 appelle la fonction `MPI_Recv` et réceptionne le message.

Ces deux fonctions existent également en communication asynchrone :

— `MPI_Isend(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator, MPI_Request *request)`

— `MPI_Irecv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Request *request)`

Elles sont semblables à leurs homologues synchrones, sauf qu'elles possèdent un argument supplémentaire : `MPI_Request *request`, il s'agit d'une mémoire tampon pour les requêtes. Celui-ci pourra être donné à la fonction `Wait(MPI_Request *request)` qui sera bloquante jusqu'à ce que les messages soient bien réceptionnés ou envoyés.
En tant qu'expéditeur, il faudra être très attentif de ne pas modifier la valeur qui est pointée depuis l'adresse mémoire tant que le message n'est pas envoyé. Sinon le résultat est indéfini.

La fonction `MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status* status)` est proche de la fonction recevoir sauf qu'elle ne va pas recevoir les données. C'est une fonction bloquante qui va attendre un message possédant un expéditeur et une balise identiques à ses paramètres. Dès que ces conditions sont respectées, elle va remplir le statut. Ensuite, le programme pourra employer un MPI_Recv pour recevoir le message.

5.1.2 Messages collectifs

Dans la section précédente, il était toujours question de messages envoyés par un nœud et réceptionnés par un autre. Cependant, une autre catégorie de messages existe : les messages collectifs destinés à tout un communicateur.

Par défaut, le communicateur englobe tous les nœuds lancés au démarrage de l'application. Il y a moyen de créer un nouveau communicateur qui ne concerne que certains nœuds avec la fonction `MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm)`. Celle-ci prend en argument, le communicateur actuel, la couleur du nœud, une clé et le nouveau communicateur. Les nœuds qui possèdent la même couleur seront dans le même communicateur et auront un numéro unique qui leur sera attribué en fonction de la valeur de la clé. Un communicateur peut être vu comme un groupe de nœuds. Toutefois, un nœud peut employer plusieurs communicateurs en même temps.

Les fonctions collectives employées dans la suite sont les suivantes :

— `MPI_Barrier(MPI_Comm communicator)` : mise en place d'une barrière pour tous les nœuds du communicateur. Une barrière est un point dans le code où les processus les plus rapides devront attendre les plus lents. Dès que tous les processus de ce groupe ont atteint la barrière, ils peuvent continuer leur exécution.

— `MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)` : un nœud va envoyer à tous les autres nœuds du communicateur la même valeur. Cette fonction est beaucoup plus efficace que d'envoyer la même valeur séparément à chaque nœud. La fonction broadcast est bloquante.

— `MPI_Allreduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator)` : chaque nœud au sein d'un communicateur va envoyer une valeur (`send_data`) et va recevoir le résultat (`recv_data`) de l'opération (`op`) appliquée sur toutes les données de tous les nœuds. Les opérations disponibles sont multiples : somme, minimum, maximum ... Cette fonction bloquante permet de trouver très rapidement le résultat de l'opération.

5.2 Adaptation de SFlow2D

L'adaptation du programme SFlow2D se base sur trois modifications clés. La première concerne l'initialisation du domaine qui permet à chaque nœud du cluster d'obtenir les informations de sa région. La seconde phase est le calcul d'un pas de temps commun pour garantir la continuité du domaine. La dernière phase est le partage et la création des fichiers de résultat de la simulation.

La figure 5.1 synthétise les grandes étapes du programme adaptées pour MPI. Elles sont exposées par la suite dans le point 5.2.2. Le programme est lancé sur un certain nombre de nœuds en même temps :

— Le nœud principal, appelé nœud master, est responsable de l'initialisation du domaine et de la création des fichiers de sorties.

— Les autres nœuds, ont chacun une région du domaine à calculer, dans la suite ils seront appelés nœuds de calcul.

Il faut donc autant de régions que de nœuds de calcul. Le programme a besoin de connaître la répartition des cellules entre les régions dans un fichier d'entrée. Par exemple, si un domaine possède 4 régions, il faudra employer 5 nœuds du cluster : un pour le master et 4 pour les nœuds de calcul.

5.2.1 Division du domaine

La division du domaine et des informations sont des points primordiaux à comprendre avant d'aborder les différentes étapes clés de l'exécution du programme. Dans le domaine de la figure 5.2, il y a deux régions : la rouge et la bleu. Chaque nœud de calcul ne possède que les nœuds, cellules et interfaces qui lui sont utiles. Cependant, les régions ont besoin des cellules qui font partie de

leur domaine ainsi que les cellules adjacentes à ce dernier. Dans notre exemple, chaque région possède les cellules qui sont marquées par un rond de sa couleur accompagnée de son numéro de cellule.

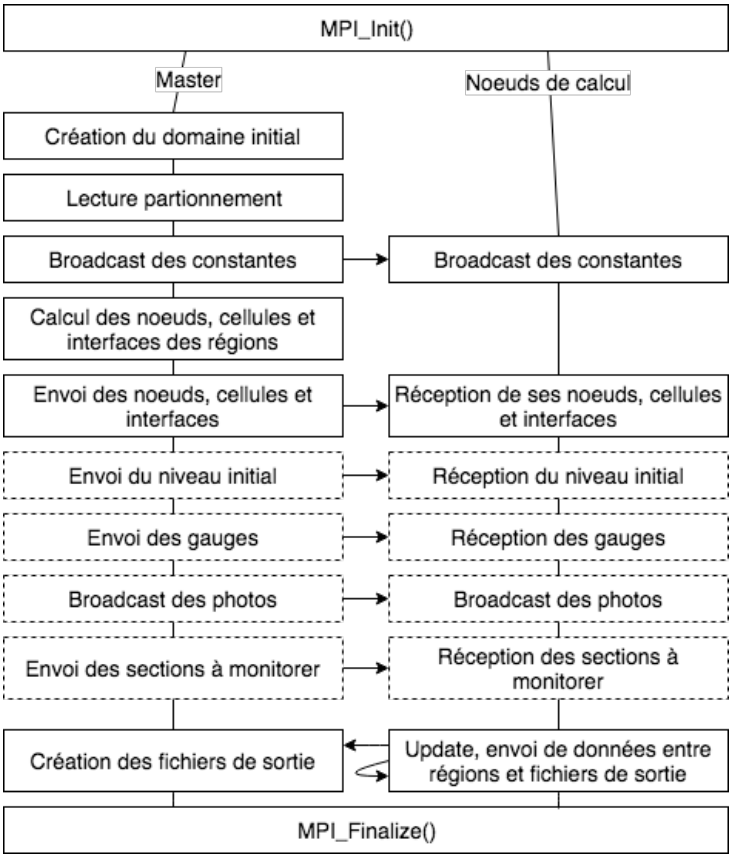


FIGURE 5.1 – Étapes du calcul et de l'envoi d'échange de données par MPI

Chaque région possède toutes les interfaces qui sont adjacentes à ses cellules initiales, mais pas celles des cellules adjacentes à son domaine. Donc le nœud de calcul qui possède la région bleue aura 8 cellules et 13 interfaces : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 15].

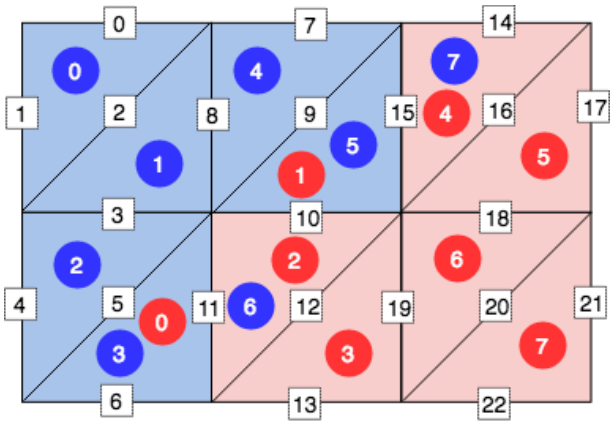


FIGURE 5.2 – Domaine d'exemple

Chaque région possède également tous les nœuds qui constituent les cellules qu'elles possèdent en mémoire.

5.2.2 Initialisation

Au démarrage du programme, le nœud principal est le seul à posséder toutes les informations du domaine. Grâce à ces informations, il peut calculer au mieux ce dont chaque région aura besoin. Les différentes étapes sont reprises ci-dessous :

1. Envoi à tous les nœuds de calcul des différents paramètres clés tels que le temps de départ, le temps de terminaison, la profondeur d'eau minimale... La fonction `broadcast` est employée.
2. Parcours l'ensemble des interfaces du domaine et enregistre les voisins de chaque région.
3. Création d'un tableau avec la correspondance entre la numérotation des cellules de chaque région et celle du master. En effet, lors de la génération des fichiers de résultats, il faut savoir retrouver l'ordre initial.
4. Parcours du fichier initial des interfaces pour faire le lien avec les nœuds dès qu'il sait exactement quelles interfaces appartiennent à quelle(s) région(s). Il sélectionne ensuite pour chaque région les nœuds qui lui appartiennent ainsi que ceux qui constituent les cellules voisines à cette région.
5. Envoie à chaque nœud de calcul le nombre de nœuds qu'il va recevoir.
6. Parcours du fichier des coordonnées des nœuds et envoi de ces informations aux nœuds de calcul qui en ont besoin.
7. Envoi du nombre de cellules à chaque région.
8. Parcours du fichier des cellules et envoi des cellules nécessaires à chaque région. Pour envoyer une cellule, le master traduit les numéros des nœuds en fonction des nœuds connus par chaque région.
9. Envoi du nombre d'interfaces pour chaque région.
10. Lecture du fichier d'interface et envoi à chaque région de leurs interfaces. Les numéros des nœuds et cellules sont transformés en fonction des informations de chaque région. S'il s'agit d'une interface spéciale (par exemple avec un flux d'eau imposé), les informations sont transmises au même moment. En parallèle à cet envoi, il y a de nombreuses informations qui sont calculées et enregistrées par le master lorsqu'il s'agit d'une interface commune à deux régions. Les informations enregistrées sont explicitées plus loin dans ce chapitre.
11. Envoi du nombre d'interfaces voisines à une autre région.
12. Envoi des informations concernant les interfaces entre deux régions.
13. Envoi du niveau de fond d'eau si l'information a été passée en entrée au programme (le drapeau zb est à 1).
14. Les informations relevées par les jauges de mesures sont envoyées si présentes.
15. Si des instantanés doivent être réalisées, un broadcast est employé pour les envoyer à toutes les régions (puisqu'il s'agit d'instantanés de tout le domaine, toutes les régions sont concernées par ces informations initiales).
16. S'il y a des sections à surveiller, les pas de temps leur sont envoyés.

Pour les interfaces qui concernent plusieurs régions, il faut de nombreuses informations pour que la transmission des informations se passe au mieux. Dans notre exemple de départ, les interfaces [11, 10 et 15] sont communes à deux régions. Seule une des deux régions doit calculer le flux traversant cette interface et donnera ce résultat à la région voisine.

Pour effectuer ces échanges correctement chaque région doit posséder pour chaque interface les informations suivantes :

— Son numéro d'interface
— Son numéro de la cellule droite et gauche
— Le numéro de la cellule droite et gauche pour la région voisine
— L'identifiant de la région voisine
— Un indicateur pour savoir si elle doit calculer cette interface ou si elle la recevra de la région voisine.

La séparation des données en fonction des régions et l'envoi de celles-ci aux différents nœuds de calcul est une étape rapide du calcul. Lai et Khan [9] et Sanders et al. [10] proposaient d'enregistrer ces données dans des fichiers séparés pour être ensuite envoyés et lus par les différents nœuds de calcul. Cette technique engendre la création de nombreux documents et est utile si une même simulation est exécutée plusieurs fois. Par souci de facilité et au vu de la rapidité de l'initialisation des régions, les données sont directement envoyées aux nœuds correspondant sans passer par des fichiers de sauvegarde.

5.2.3 Itérations

Le fonctionnement des itérations n'est pas fondamentalement différent de ce qui a été présenté dans le chapitre 1. La suite présente les modifications apportées pour permettre la synchronisation des données entre les différents nœuds du cluster. La figure 5.4 présente les différentes étapes d'une itération.

Pour trouver le pas de temps minimal commun à toutes les régions, la fonction `MPI_Allreduce` conjointement à l'opération minimum. Ce qui signifie que tous les nœuds vont envoyer leur pas de temps minimal et recevront en retour le pas de temps minimal envoyé par un des nœuds. Le communicateur employé comprend tous les nœuds de calcul (donc pas le master). Puisque cette fonction est bloquante, elle permet aussi à s'assurer que tous les nœuds calculent bien la même itération au même moment (étape 2).

Lors du calcul d'une interface, en fonction des valeurs des deux cellules adjacentes, un flux partant vers ces dernières est calculé. La figure 5.3 illustre la suite des explications.

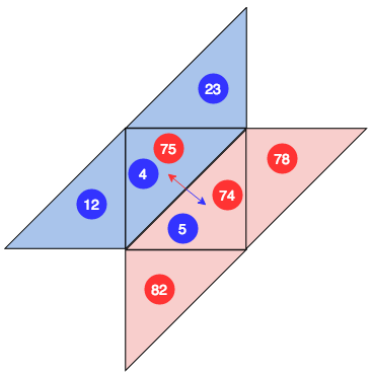


FIGURE 5.3 – Échanges d'information à l'interface de 2 régions

À l'envoi des données de départ, le master choisit la région responsable de l'interface, pour l'exemple, la région bleue sera la responsable. La région bleue possède les informations à jour des cellules 4 et 5. Grâce à cela, elle peut calculer le flux d'eau qui va aller dans les cellules 4 et 5. Après avoir calculé le flux de toutes les interfaces frontière dont elle est responsable (étape 4), la région bleue va envoyer à la région rouge le buffer de la cellule 5 et lui donne également le

numéro correspondant à la région de destination (donc la cellule 74) (étape 5). La région rouge va donc ajouter au buffer de flux de sa cellule 74 la valeur reçue du bleu (étape 7).

La seconde étape est le calcul des interfaces qui ne sont pas frontière (étape 6). La région rouge va calculer, entre autres, les flux qui vont des cellules 78 et 82 vers la cellule 74.

La troisième étape est la mise à jour des cellules (étape 9). La région rouge va calculer les nouvelles valeurs pour ses cellules, dont la 74 et va ensuite envoyer les dernières valeurs de cette cellule à la région bleue (étape 10). Grâce à cela, la région bleue pourra calculer le flux d'eau lors de la prochaine itération avec des valeurs à jour (étape 11).

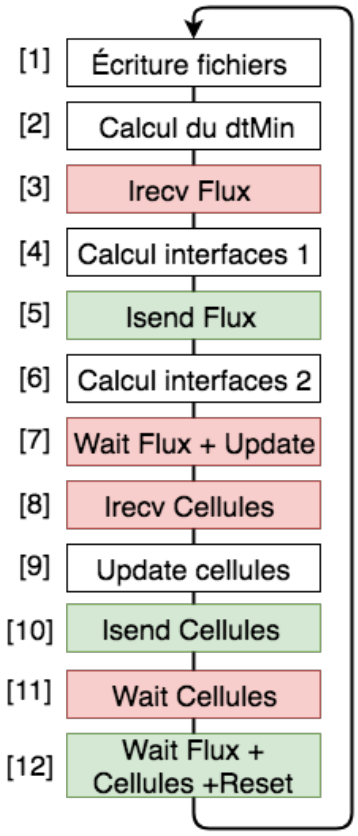


FIGURE 5.4 – Schéma itératif d'un nœud du cluster

Sur le schéma, les étapes en rouge correspondent aux éléments à recevoir :

— L'étape 3 indique l'allocation de mémoire pour les flux à recevoir des régions voisines.

— L'étape 7 attend d'avoir reçu le flux pour une certaine cellule puis la met à jour.

— L'étape 8 alloue de la mémoire pour la cellule mise à jour à recevoir.

— L'étape 11 s'assure que toutes ces cellules ont été reçues.

Les étapes en vert, quant à elles, indiquent les éléments à envoyer :

— L'étape 5 et 6 envoient respectivement le flux et les données des cellules.

— L'étape 12 s'assure que les flux et cellules ont été correctement envoyés et réceptionnés. Ensuite, les buffers temporaires sont remis à zéro.

Comme leurs noms l'indiquent, et vu la présence de `Wait`, les messages sont asynchrones pour éviter d'être bloqués par un nœud de calcul qui prend plus de temps lors de certaines étapes.

Les buffers de toutes les cellules sont remis à zéro à chaque itération. Et pour éviter d'envoyer plusieurs fois un même buffer, une région calcule d'abord toutes les interfaces frontière (étape 4) puis envoie les buffers pour chaque cellule extérieure à la frontière (étape 5). En se basant sur l'exemple initial de la figure 5.2, si la région bleue est responsable de l'interface 10 et 11, il va sommer les deux flux dans le buffer de la cellule 6. Ensuite, il l'enverra une unique fois à la région rouge en lui disant que c'est sa cellule 2.

5.2.4 Fichiers de sortie

Le nœud master, après avoir distribué les informations de départ à chaque nœud de calcul, sera responsable de la génération des fichiers de sortie.
Il effectue une boucle infinie dans laquelle il exécute la fonction `Probe` pour voir le prochain type de fichier qu'il devra générer. S'il reçoit un message qui comporte le type *stop*, il sait qu'il doit s'arrêter. Ce type de message est envoyé par le nœud de calcul n°1 à la fin de la simulation.

Le type du message est particulier : pour éviter de confondre les messages lors de la génération des fichiers (par exemple si le message de l'itération X+1 arrive avant la X), le type du message inclut le type de fichier à générer ainsi que le numéro du fichier. Ce numéro est incrémenté à chaque nouveau fichier. Puisque le `Send` prend comme type un nombre entier de 32 bits, les deux nombres cités précédemment seront assemblés dans cet entier. En effet, le type des messages est inférieur à 16, il lui faut donc 4 bits. Et le compteur du numéro de fichier a été fixé arbitrairement à 8 bits, il est donc compris entre 0 et 255. Si le nombre est supérieur, un modulo lui est appliqué. Le type du fichier à générer est un chiffre allant de 2 à 9, pour chaque type de fichier, deux chiffres sont attribués, un premier pour le temps et un second pour les données. Les deux nombres de 4 et 8 bits formant le type sont rassemblés pour former un nombre unique de 12 bits présenté à la figure 5.5.

FIGURE 5.5 – Format du type du message transportant les informations pour la génération des fichiers de sortie

Pour en revenir à la fonction `Probe`, le master regarde les 4 bits de poids fort (ceux à gauche) pour voir de quel type de message il s'agit. Ensuite, il appelle la fonction correspondant à ce type. Cette fonction garde un compteur interne et sait donc exactement de qui elle doit recevoir un message et avec quel nombre unique. Grâce à ces deux nombres, il n'y a pas de mélange de messages possible.

Un des nœuds de calcul est responsable d'envoyer le temps actuel de la simulation, et tous les nœuds envoient leurs informations en un message pour la génération d'un fichier. Pour l'envoi, une nouvelle structure a été créée : un entier accompagné de trois nombres `double`. Ce type permet d'envoyer le numéro de la cellule avec ses variables hydrauliques. Le master connaît la numérotation des cellules de chaque région ce qui lui permet de les réordonner. Lorsqu'il possède toutes les informations pour un fichier, elle l'enregistre.

5.3 Résultats

Ce programme a été testé sur le cluster **Lemaitre3** du CÉCI avec le cas Toce et Tous. Le cluster **Lemaitre3** est composé de 80 serveurs caractérisés par 2 processeurs à 12 cœurs Intel Skylake 5118 cadencés à 2,3 GHz. Avec le Turbo Boost, il peut atteindre une fréquence de

3,2 GHz. Chaque serveur possède 96 Gb de RAM. Il est important de noter que le CÉCI met à disposition des processeurs plus performants que le serveur sur lequel la version multithread a été exécutée.
"Les moyens de calcul ont été fournis par le Consortium des Équipements de Calcul Intensif (CÉCI), financé par les Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) sous la convention n° 2.5020.11".

Le partitionnement des cellules est fourni en entrée, car cette étape est considérée comme du précalcul et n'est donc pas comptée dans les temps d'exécution. De plus, tous les résultats ont été vérifiés avec la simulation de base suivant la méthodologie décrite au chapitre 2.

5.3.1 Toce

Pour le Toce, deux métriques de distance ont été utilisées : la matrice d'adjacence et la distance euclidienne.

Chaque mesure a été exécutée 10 fois sur le calcul pour différentes tailles de partitionnement à savoir 4, 8, 23, 24, 47, 48, 95 et 96 régions. Les résultats pour l'usage de la distance euclidienne sont également indiqués dans la table 5.1. Toutefois, les résultats sont présentés avec un code qui n'a pas été optimisé par le compilateur au vu de la petite taille du domaine et de la rapidité d'exécution.

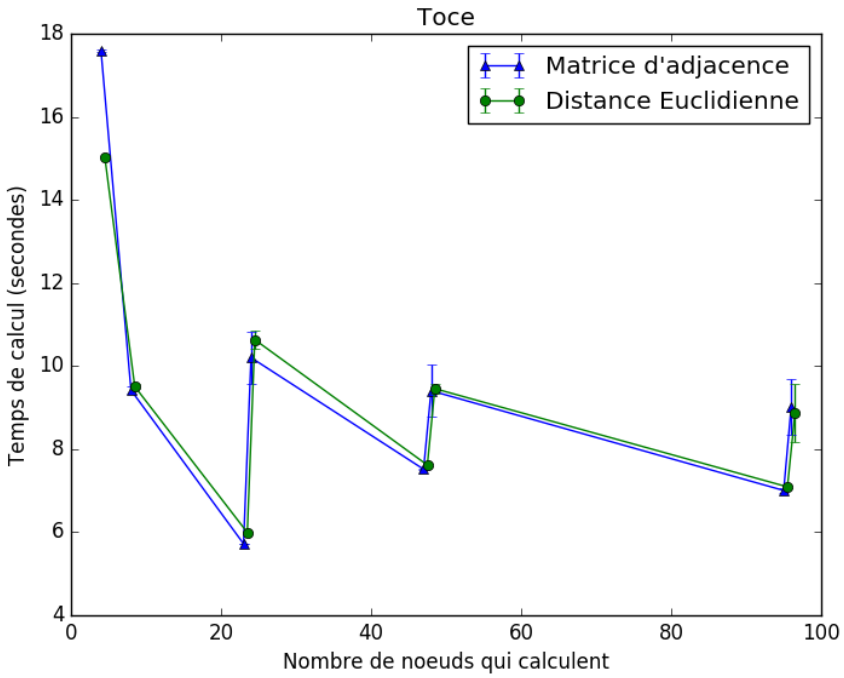


FIGURE 5.6 – Graphe du temps de calcul par rapport au nombre de nœuds pour Toce

L'analyse du graphique 5.6 permet de voir l'amélioration du temps de calcul en fonction du nombre de nœuds. L'amélioration est constante pour les nœuds, 4, 8 et 23. À ce dernier nœud, un minimum est atteint. Si le nombre de nœuds augmente, le temps sera supérieur ! En effet, l'augmentation des transferts d'informations entre nœuds qui ne sont pas compensés par le gain du temps des calculs dû à la diminution de la taille des régions.

L'augmentation du passage de 23 à 24 nœuds de calcul s'explique par la topologie du cluster : chaque ordinateur possède 24 cœurs. En effet, pour le 24 nœuds de calcul, il a été nécessaire

d'obtenir 25 nœuds (1 pour le master et 24 pour les calculs), il a donc fallu 2 serveurs différents. Cette différence fait augmenter le temps de transfert des messages et donc celui du calcul total. Cette même différence est également présente lors du passage de 47 à 48 nœuds de calcul et celui de 95 à 96.

Un fait étrange peut être observé avec 4 nœuds : le calcul est plus rapide avec le partitionnement employant les distances euclidiennes que la matrice d'adjacence. Pourtant cela devrait être le contraire. En étudiant plus en détail les deux partitionnements présentés au chapitre 3, la différence de taille des frontières est importante. Le partitionnement avec la matrice des distances fournit trois régions avec relativement peu de frontières et une région centrale en possédant un nombre élevé : 140. En opposition au nombre maximal de frontières pour le partitionnement avec la distance euclidienne qui est de 100. Malgré le partitionnement équitable en termes de cellules par région, cette différence de frontière maximale se fait ressentir au niveau des résultats.

Pour toutes les autres tailles de régions, le nombre de frontières maximal est toujours meilleur avec la matrice d'adjacence (pour 8 régions : 95 vs 115 ; pour 48 régions : 52 vs 99 ; pour 96 régions : 40 vs 61).

Version	**Temps d'exécution**	**Speed-up**	**Efficacité**
de base			
de base -O0	73,62 s		
MPI -O0 et distance euclidienne			
4 (+1)	15,04 s	x 4,49	0,979
8 (+1)	9,51 s	x 7,74	0,86
23 (+1)	5,99 s	x 12,29	0,512
24 (+1)	10,63 s	x 6,92	0,277

TABLE 5.1 – Temps d'exécution de Toce avec la distance euclidienne avec -O0

Dans la table 5.1, la version de base a été exécutée sur le cluster avec un seul nœud. L'efficacité calculée est très intéressante pour un petit nombre de nœuds, elle a été calculée en tenant compte du nœud master. Dans la première colonne, le nombre entre parenthèses correspond au nœud master.

5.3.2 Tous

La figure 5.7 montre les résultats de la simulation du cas Tous. Pour rappel, il s'agit d'un grand domaine qui compte plus de 60 000 cellules. Cette simulation prenait 2 jours et 6 heures à calculer sur un ordinateur standard avec le programme de base avec l'optimisation du compilateur. Grâce au programme parallélisé pour fonctionner sur un cluster, moins d'une heure est suffisante pour boucler ce même travail. Le code de base a également été lancé sur une seule unité de calcul du cluster, les résultats, en comparaison avec les sorties du code de MPI, sont présentés à la table 5.2. Le code de base a été lancé sur un seul nœud, et si le serveur était peu employé à ce moment-là, le processeur peut activer le Turbo Boost qui augmente significativement sa fréquence. Le Turbo Boost s'active dans certaines situations qui dépendent, entre autres, du nombre de cœurs actifs, de la charge de travail ou encore de la température du processeur [33].

Plus le nombre de nœuds de calcul augmente, plus le temps de calcul diminue. L'amélioration est intéressante jusqu'à 47 nœuds, au-delà, l'utilisation de nœuds supplémentaires ne se justifie plus, car le gain de temps est minime.

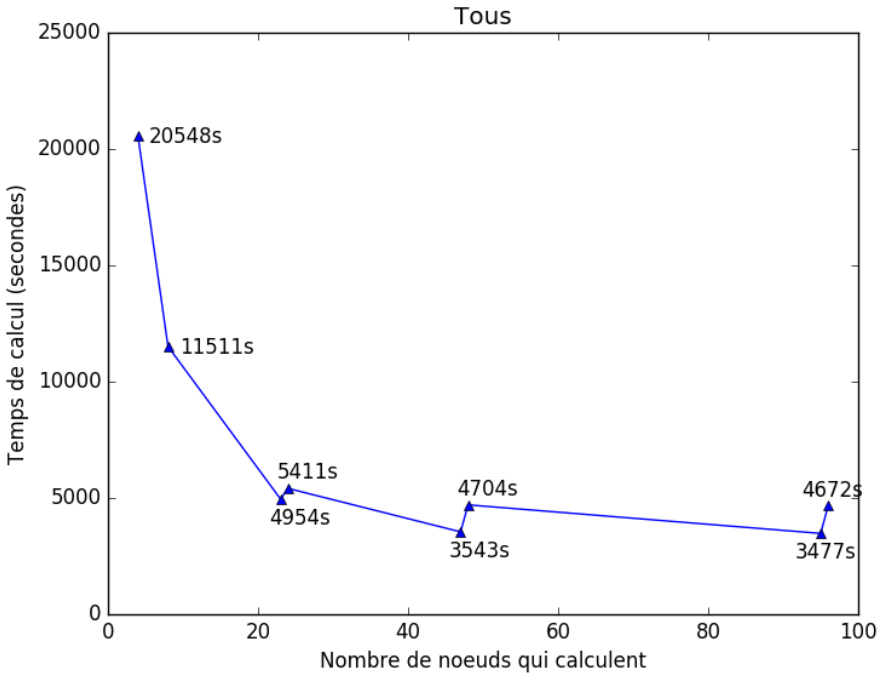


FIGURE 5.7 – Graphe du temps de calcul par rapport au nombre de nœuds pour Tous

Dans la table 5.2, les différentes tailles des nœuds de calcul employés sont reprises. Pour chacun d'eux, le gain et l'efficacité ont été calculés par rapport au programme de base avec et sans optimisation. Dans la troisième colonne, le gain et l'efficacité ont été calculés en comparaison au programme de base fonctionnant sur le cluster sans optimisation du compilateur. La dernière colonne est similaire du point de vue du contenu, mais la comparaison est réalisée sur le programme optimisé par le compilateur. Le gain a été calculé en tenant compte du nœud master qui s'occupe de l'écriture des fichiers. Le programme de MPI a été compilé avec le drapeau d'optimisation -O3 et la distance euclidienne a été employée pour diviser le domaine.

En fonction du domaine et de la topologie du cluster, il est intéressant d'allouer plus ou moins

Version	**Temps d'exécution**	**Speed-up -O0 (efficacité)**	**Speed-up -O3 (efficacité)**
de base			
de base -O0	26h 42m 14,5s		
de base -O3	16h 48m 8,3s	x 1,59 (1,0)	
MPI -O3 et distance euclidienne			
4 (+1)	5h 42m 28,2s	x 4,68 (0,94)	x 2,94 (0,59)
8 (+1)	3h 11m 50,8s	x 8,35 (0,93)	x 5,25 (0,58)
23 (+1)	1h 22m 33,9s	x 19,41 (0,81)	x 12,21 (0,51)
24 (+1)	1h 30m 11,5s	x 17,76 (0,71)	x 11,18 (0,45)
47 (+1)	0h 59m 3,1s	x 27,13 (0,57)	x 17,07 (0,36)
48 (+1)	1h 18m 24s	x 20,44 (0,42)	x 12,86 (0,26)
95 (+1)	0h 57m 57,2s	x 27,65 (0,29)	x 17,4 (0,18)
96 (+1)	1h 17m 52,4s	x 20,58 (0,21)	x 12,95 (0,13)

TABLE 5.2 – Temps d'exécution de Tous avec la distance euclidienne avec -O3

de nœuds. Pour obtenir un gain de temps optimal, il faut trouver un compromis entre la taille des régions et la répartition physique des nœuds de calcul au sein du cluster. Par exemple pour le Toce, il valait mieux rester sur un même serveur qui contenait tous les nœuds. Par contre pour le Tous, puisqu'il est composé de nombreuses cellules, il est intéressant de posséder beaucoup de nœuds répartis sur plusieurs serveurs du cluster. Il ne faut pas non plus oublier le temps nécessaire à la décomposition du domaine en régions qui est plus élevé lorsque le nombre de régions augmente.

L'occupation du cluster est également un aspect important à prendre en compte. En effet, les machines du cluster sont généralement employées par un nombre important de personnes et le scheduler essaie de gérer au mieux les différents travaux qui lui sont envoyés. Parfois, il est plus intéressant de demander un nombre moins important de nœuds pour que le scheduler nous trouve plus rapidement une place et donc le temps passé dans la file d'attente est diminué.

6

Programmation parallèle sur Processeur Graphique

Ce chapitre aborde succinctement la différence entre le CPU et le GPU et le fonctionnement du GPU. Ensuite, une comparaison entre les différentes solutions de programmation sur GPU, ainsi que les choix d'implémentation seront présentés. Finalement, le chapitre s'attaquera aux résultats de performance de l'implémentation.

6.1 Architecture générale du GPU

Le calcul accéléré sur GPU, General-Purpose GPU ou GPGPU, consiste en l'utilisation de processeur graphique (GPU) avec un processeur (CPU) afin d'accélérer les codes de calculs scientifiques. Les GPUs possèdent typiquement une architecture significativement différente du CPU traditionnel. Les architectures d'un seul CPU et d'un GPU NVIDIA sont montrées sur l'image 6.1.

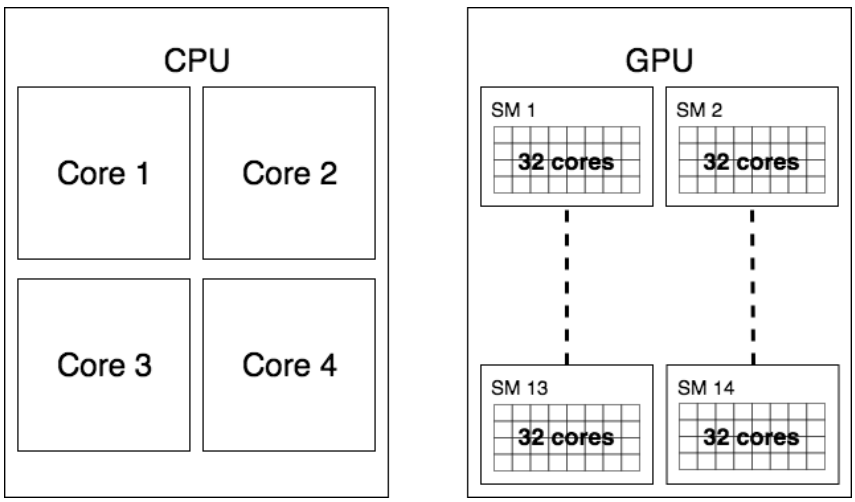


FIGURE 6.1 – Architecture simplifiée du CPU et du GPU NVIDIA

Le processeur moderne est composé de plusieurs cœurs, chacun optimisé pour le traitement séquentiel, tandis que le GPU NVIDIA a une architecture massivement parallèle constituée de Streaming Multiprocessors (SM sur la figure 6.1) qui sont à leur tour subdivisés en de plus petits cœurs, les cœurs de calculs CUDA (un carré sur la figure 6.1). Ces plus petits cœurs plus efficaces sont conçus pour la manipulation de multiples tâches simultanées. Si ces milliers de tâches s'exécutent en parallèle, une énorme puissance de calcul combinée à une grande bande passante mémoire peuvent être mise à profit ; toutefois, cela requiert d'écrire du code efficace [34].

6.2 Choix du framework de développement sur GPU

Une implémentation efficace sur GPU nécessite d'utiliser des outils adaptés. Ce point compare les différents langages et frameworks disponibles.

6.2.1 Les Frameworks

Les frameworks de programmation parallèle les plus largement utilisés sur des systèmes hétérogènes incluent OpenACC[35], OpenCL[36], OpenMP[26], et NVIDIA CUDA[37]. Ces 4 frameworks se répartissent dans 2 catégories : les frameworks basés sur des directives et les Interfaces de programmation bas-niveau.

	OpenACC	OpenMP	OpenCL	CUDA
Parallélisme	- Parallélisme des données - Parallélisme de tâches asynchrones - GPU uniquement	- Parallélisme des données - Parallélisme de tâches asynchrones - CPU et GPU	- Parallélisme des données - Parallélisme de tâches asynchrones - CPU et GPU	- Parallélisme des données - Parallélisme de tâches asynchrones - GPU uniquement
Abstraction de l'architecture	- Hiérarchie de la mémoire - Correspondance et mouvement explicite des données	- Hiérarchie de la mémoire - Liaison de données et de calculs - Correspondance et mouvement explicite des données	- Hiérarchie de la mémoire - Correspondance et mouvement explicite des données	- Hiérarchie de la mémoire - Correspondance et mouvement explicite des données
Synchronisation	- Réduction - Join	- Barrière - Réduction - Join	- Barrière - Réduction	- Barrière
Implémentation framework	Directives pour le compilateur en C/C++ et Fortran	Directives pour le compilateur en C/C++ et Fortran	Extension de C/C++	Extension de C/C++

Table 6.1 – Propriétés principales d'OpenACC, OpenMP, OpenCL, et CUDA

Le tableau 6.1 résume les principales propriétés des langages de programmation parallèles. OpenMP et OpenACC sont essentiellement implémentés au travers de *directives pour le compilateur* en C/C++ et Fortran, ce qui permet de cacher les détails d'architecture du programmeur. Tandis que OpenCL et CUDA sont implémentés tels des libraires en C et C++ et expose le programmeur à des détails d'architecture bas-niveau.
Concernant le support de parallélisme, tous les frameworks supportent le parallélisme des données et de tâches asynchrones. Alors qu'OpenMP et OpenCL offrent tous deux de la parallélisation sur CPUs *hôtes* multicœurs et sur dispositifs pluri-cœurs, OpenACC et CUDA proposent des outils de parallélisation uniquement sur des accélérateurs tels que les GPUs NVIDIA.

6.2.2 Langage de programmation

Si OpenACC et OpenMP offrent une simplicité de programmation au programmeur, les programmeurs se tournent vers OpenCL et CUDA afin d'augmenter au plus les performances de tâches dont la granularité de parallélisme est très fine et d'optimiser les accès mémoire.
Le choix de CUDA comme langage de programmation s'est fait sur base des quatre arguments suivants :

1. La fine granularité de la parallélisation.
2. L'optimisation des accès à la mémoire.
3. La popularité au sein de la communauté scientifique de CUDA comme langage de programmation parallèle.

4. Des métriques de performance comparant OpenACC, OpenMP, OpenCL, et CUDA dans l'article *Benchmarking OpenCL, OpenACC, OpenMP, and CUDA : programming productivity, performance, and energy consumption* [6].

6.3 Implémentation sur GPU

Les performances des applications accélérées sur GPU par le biais de CUDA se justifient par le fonctionnement de CUDA et l'architecture des GPUs NVIDIA.

6.3.1 Fonctionnement de CUDA

L'élément de calcul de base, similaire au CPU, est le *thread*. Un groupe de 32 threads est dénommé un *warp*. Les threads, dans l'environnement CUDA, sont identifiés par un index qui varie de `0` à `blocDim`, la dimension d'un bloc ou autrement dit le nombre de threads regroupés dans un bloc.

Le GPU attribue chaque bloc à son Streaming Multiprocessor. Chaque Streaming Multiprocessor a un multiple de 32 cœurs de calcul CUDA et attribue dès lors les threads de son bloc à chacun de ses cœurs de calcul. Tous les cœurs de calcul d'un Streaming Multiprocessor effectuent les mêmes opérations en même temps, mais appliquées chacun à un élément différent au sein d'un vecteur.

Par le fait que le nombre de cœurs de calcul d'un Streaming Multiprocessor est un multiple de 32, il est possible d'y lancer plusieurs warps ou plusieurs fois 32 threads en parallèle. Il est donc recommandé d'attribuer une taille de bloc multiple de 32 [37]. Pour gérer une telle quantité de threads, le GPU utilise une architecture unique appelée SIMT (Single-Instruction, Multiple-Thread). Cette architecture permet la parallélisation au niveau de l'exécution des instructions sur le processeur.

6.3.2 Modèle de Programmation CUDA

CUDA C est une extension du langage de programmation C qui inclut la possibilité de spécifier 3 abstractions clés lors d'une exécution : la hiérarchie des threads, la hiérarchie de la mémoire et les barrières de synchronisation.[37, 5] Ces 3 abstractions se caractérisent par l'introduction d'extensions dans le code.

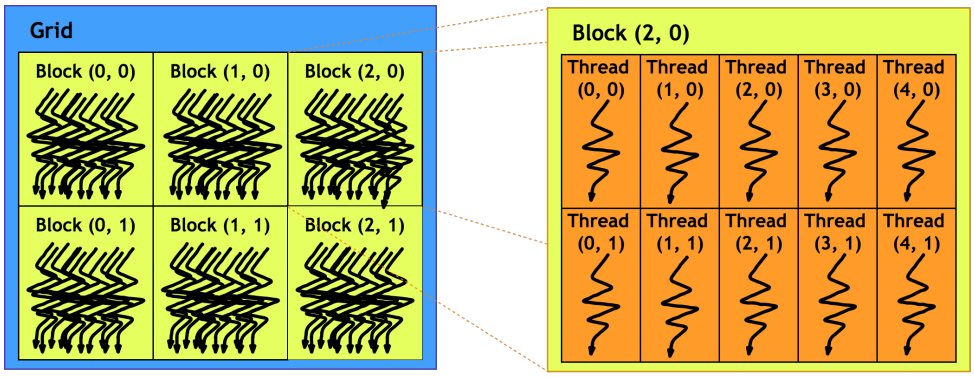


FIGURE 6.2 – Grille de blocs de threads

Hiérarchie des threads

Le changement le plus significatif se situe au niveau de la configuration de l'appel de fonction où la hiérarchie des threads peut être spécifiée. Un appel de fonction ou lancement de kernel se fait de la façon suivante :

```
maFonction
<<<
    blocsParGrille, // taille de la grille ou le nombre de blocs exécutés
    threadsParBloc, // taille d'un bloc ou le nombre de threads dans le bloc
    ...             // arguments supplémentaires
>>>(parametres);    // paramètres de la fonction
```

Code Source 6.1 – Kernel parameters

La configuration de l'exécution d'un kernel est dépendante de la carte graphique sur laquelle il est exécuté. Le nombre de `blocksParGrille` suit la relation 6.1.

$$\texttt{blocksParGrille} = \frac{N + \textit{ taille d'un bloc } - 1}{\textit{taille d'un bloc}} \tag{6.1}$$

Le point 6.4 explique par un exemple, l'importance de la configuration de l'exécution d'un kernel sur un GPU.

Hiérarchie de la mémoire et barrière d'exécution

Les threads CUDA ont également accès à de multiples espaces mémoire durant leur exécution. Chaque thread a sa propre mémoire locale (`idx`). Chaque bloc de threads a une mémoire partagée (`s_mem`) accessible à tous les threads d'un bloc et dont la durée de vie est identique à celle d'un bloc. Tous les threads ont accès à la mémoire globale. Il existe deux autres types de mémoire, mais ils sortent du cadre de ce mémoire. Lorsque le kernel `maFonction` est lancé, chaque thread exécute ses opérations, crée ses propres variables, accède à la mémoire partagée ou globale, et s'arrête à la barrière de synchronisation et attend que tous les autres threads du même bloc atteignent cette barrière afin de reprendre l'exécution du kernel.

```
__global__ void maFonction(parametres){
    int idx; // index du thread
    // mémoire externe partagée allouée dynamiquement
    extern __shared__ int e_smem[];
    // mémoire partagée allouée statiquement
    __shared__ int s_mem[nombre_elements];
    ... // calculs sur les mémoires partagées
    __synchthreads(); //barrière de synchronisation des threads d'un bloc
}
int main() {
    int N = 1 << 20;
    float * d_mem, * u_g_mem;;
    cudaMalloc( (void**)&(d_mem), N * sizeof(float) );
    ...
    maFonction<<< blocsParGrille, threadsParBloc, Ns, streamID>>>(d_mem);
    ...
}
```

Code Source 6.2 – Exécution d'un kernel

Un programme exécuté sur le GPU est performant lorsqu'il fait énormément d'appels aux mêmes kernels qui minimisent les accès en mémoire globale et profitent au mieux de la mémoire partagée au sein d'un même bloc. Un programme accédant à la mémoire fréquemment ne sera pas plus efficace s'il est implémenté sur un GPU plutôt qu'un CPU. Au contraire, la limitation au niveau de la bande passante de la mémoire globale du GPU, fait que ce programme sera plus lent.

Suivant ce modèle de programmation, le développeur doit lancer le code CPU et initialiser les données, transférer les données sur le GPU, exécuter le kernel, retransférer les données vers le CPU et finaliser l'exécution du code CPU. Dans un souci d'unification des accès entre la mémoire du CPU et la mémoire du GPU, CUDA a introduit la notion d'Unified Memory Access (UMA) avec leur kit de développement CUDA 6 [38, 39]. Le développeur n'a ainsi plus besoin de s'occuper des détails de transfert de mémoire entre CPU et GPU. La mémoire unifiée nécessite cependant l'utilisant d'un GPU de classe Kepler ou plus récente et l'utilisation d'un système d'exploitation non embarqué (Linux, Windows, macOS).

Le tableau 6.2 présente les avantages et les inconvénients liés à l'utilisation de l'accès unifié à la mémoire remplaçant l'allocation de mémoire GPU et de la copie des données de la mémoire du CPU vers celle du GPU. L'un des enjeux les plus importants de l'UMA est le concept de cache miss. Lorsque le programme tente d'accéder à une donnée et si elle n'est pas présente en mémoire cache, cela engendre des retards dans l'exécution, car la donnée doit être récupérée, dans le cas du GPU, dans la mémoire globale à la bande passante limitée.

Avantages	**Inconvénients**
Facilité d'utilisation	
Allocation de la mémoire accessible au CPU et GPU	Temps de transfert entre mémoires CPU et GPU importants
Possibilité de gestion des transferts entre mémoire du CPU et du GPU	Cache miss
Unification des accès mémoire entre CPUs et multi-GPUs	

Table 6.2 – Avantages et inconvénients de l'accès unifié en mémoire (UMA)

Parallélisation des exécutions

CUDA définit également la possibilité de lancer les kernels sur différents fils d'exécution, les streams. Dans l'exemple 6.2, cela se définit par la variable `streamID`. Si le fil d'exécution n'est pas spécifié comme dans l'exemple 6.1, le kernel s'exécute sur le fil d'exécution principal.

Structure du domaine de calcul

Le programme SFlow2D initialement écrit en orienté objet, a dû être modifié pour s'adapter à l'architecture de la mémoire du GPU, la configuration des exécutions des kernels et pour bénéficier des performances de calcul du GPU. La nouvelle structure du domaine ne se compose plus de listes d'objets comme schématisés à la figure 6.3, mais de structures de tableaux ou tableaux de structures (voir figure 6.4).

La structure du domaine doit se faire sous forme de tableau de structures, car les calculs de flux et les mises à jour des cellules n'accèdent pas uniquement à la variable h (la hauteur de l'eau),

mais également aux variables p et q (débit selon x et y respectivement) dont les emplacements en mémoire sont contigus. Les accès mémoires profitent ainsi de la localité spatiale. La localité spatiale définit que des éléments proches en mémoire ont tendance à être référencés à des instants proches aussi. Par conséquent, à l'exécution d'un kernel et lors d'un accès à la variable h d'une cellule, les variables contiguës seront également en mémoire cache, la mémoire rapide spécifique à un cœur. Cette mémoire cache est également présente dans un CPU.

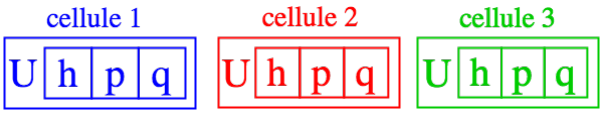


FIGURE 6.3 – Schéma simplifié des informations d'une cellule dans SFlow2D

La nouvelle structure du domaine profite donc de la spatialité temporelle tout en respectant les notations du programme initial.

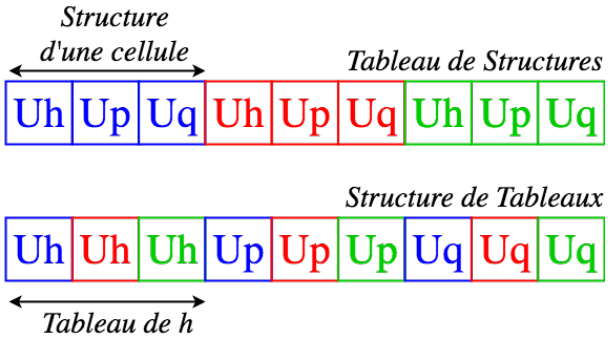


FIGURE 6.4 – Différence entre structure de tableaux et tableau de structures

Structure de l'exécution

Le schéma itératif de mise à jour du domaine se fait de la même manière que le programme initial. Le calcul des différents flux peut se faire sur plusieurs streams introduisant ainsi de la concurrence entre les flux. Le calcul du flux dans les interfaces non frontière, dont le nombre est important par rapport aux interfaces frontière, se fait sur un stream (le stream vert), tandis que les autres interfaces frontière se font sur un autre stream (le stream rouge).

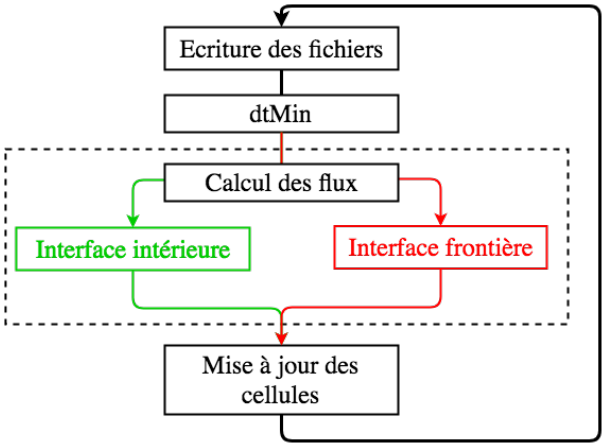


FIGURE 6.5 – Schéma itératif sur deux streams

6.4 Résultats

Avant d'analyser les résultats, il est important de noter que les temps d'exécution sur GPU sont dépendants de la configuration de l'exécution, de la version de CUDA et des caractéristiques de la carte graphique. La figure 6.3 reprend les caractéristiques du GPU sur lequel les résultats ont été générés.

Caractéristique	**Valeur**
Nombre de cœurs CUDA	1536
Nombre de Streaming Multiprocessors (SM)	8 (192 CUDA cœurs/SM)
Vitesse de la mémoire	7,0 Gb/s
Vitesse de la bande passante mémoire	224,3 Gb/s

TABLE 6.3 – Caractéristiques de la carte graphique GeForce GTX 770

Avant d'exécuter la version GPU sur une carte graphique, il faut analyser les caractéristiques de la carte graphique. Dans ce cas-ci, la GTX 770 possède 8 SM de chacun 192 cœurs de calcul CUDA. À partir du tableau 6.4 comparant les configurations d'exécution, il y a une corrélation entre le nombre de cœurs de calcul CUDA et le temps d'exécution qui apparaît. La meilleure configuration pour exécuter un kernel est de 384 threads par block et ensuite de 192 threads par block. Cela correspond au nombre de cœurs CUDA par SM ou un multiple de ce nombre. Cette démarche est importante afin de s'assurer que tous les threads sont utilisés lors de l'exécution du kernel.

La configuration de l'exécution des kernels a été générée suivant la relation 6.1 liant la taille du block au nombre de blocks.

Threads/blocks	**Cellules # blocks**	**Interfaces # blocks**	**Temps exécution (s)**
32	367	557	15,67
128	92	140	15,44
192	62	93	15,37
256	46	70	15,51
384	**31**	**47**	**15,32**
512	23	31	15,84
1024	12	18	15,59

TABLE 6.4 – Temps d'exécution moyen en fonction de la configuration Threads/Nombre de blocks

La meilleure configuration reprise en gras dans le tableau 6.4 présente une accélération de 4,80 par rapport au temps d'exécution de base c'est-à-dire de 73,62 secondes.

6.4.1 Limitations

Nous avions décidé dès le début du mémoire de paralléliser le programme SFlow2D en utilisant trois supports différents. Ce choix ambitieux, surtout pour la parallélisation sur GPU, nous a permis de produire une implémentation sur GPU de SFlow2D produisant des résultats corrects sur le cas de test Toce. Cependant, lors des tests sur le cas de test Tous, des erreurs sont apparues à 21 000 secondes sur les 144 000 secondes simulées. À partir de cet instant, les erreurs ont un effet boule de neige. Dans l'instantané produit à 15 000 secondes, les valeurs de la hauteur d'eau calculées ne diffèrent que de quelques micromètres, ce qui reste dans les bornes définies dans le chapitre 2. À partir de 21 000 secondes, les erreurs dépassent les bornes et atteignent des différences de plusieurs mètres aux environs de 40 000 secondes. Nous avons étudié le problème,

et nous avons trouvé une piste qui pourrait expliquer le problème : la précision de calcul (comme expliqué sur la page de référence [40]). Sur Toce, la durée de simulation est trop courte pour que les erreurs deviennent significatives, contrairement à Tous.

La version GPU ne garantit pas de résultats fiables pour les différentes tailles de domaine et différents temps de simulation. Nous ne considérons donc pas ce programme fiable pour de nouvelles entrées. Par contre, les GPUs récents offrent de belles perspectives d'améliorations tout en garantissant une précision accrue, il serait donc intéressant de continuer ce travail sur base de ce qui a été accompli jusqu'à présent.

6.5 Améliorations futures

La littérature suggère les pistes suivantes dans une amélioration future de la version GPU :

— L'utilisation d'un maillage structuré est avantageuse dans un GPU dû à sa configuration. Le premier avantage est la position proche des cellules en mémoire. Le second est lors du calcul des flux aux interfaces. En effet, le calcul effectué sur un maillage non structuré n'est pas la même que sur un maillage structuré. Par conséquent, une version GPU du calcul sur maillage structuré est plus rapide dû à la coalescence des accès à la mémoire globale, réduisant grandement les caches miss.

— L'article [5] propose deux améliorations supplémentaires : réordonner les cellules et réordonner les interfaces. Le nouvel ordre permet de réduire les accès disparates à la mémoire, de profiter de la localité spatiale et de maximiser la coalescence des accès. Par conséquent, il réduit les caches miss.

— D'autres sources suggèrent l'utilisation du multi-GPU, afin d'y redistribuer les calculs. La division du domaine en régions ainsi que la synchronisation entre les régions présentées dans la partie 5 combinée à la facilitée d'utilisation de l'accès unifié à la mémoire entre le CPU et les GPUs permettent d'adapter le programme SFlow2D au multi-GPU. Cependant, le coût lié à l'achat de cartes graphiques n'est pas un élément négligeable, surtout des cartes NVIDIA de la gamme **Tesla** favorisant les calculs intensifs grâce au nombre de cœurs de calculs CUDA très élevé (et donc beaucoup d'opérations en virgule flottante) et à une bande passante importante.

— L'utilisation de la mémoire de texture est une bonne alternative à la mémoire globale. Elle nécessite toutefois d'adapter le code, les structures de données en mémoire ainsi que la façon dont elles sont gérées. Cette mémoire n'est pas accessible à partir du CPU et ne peut être modifiée que dans un kernel par le GPU.

L'évolution des cartes graphiques et l'utilisation de plus en plus courante de celles-ci dans le cadre de calculs à grande échelle motivent également la continuation du développement du code.

Conclusion

Le but de ce mémoire était de rendre le code SFlow2D plus rapide par la parallélisation. L'étude préliminaire du code et des aspects physiques ont permis de comprendre les liens entre les données et les enjeux du programme. Les itérations composées du calcul du pas de temps, le calcul des flux à chaque interface et la mise à jours des données hydrauliques de chaque cellule sont des parties hautement parallélisables. Les variables de flux de chaque cellule sont susceptibles d'être accédées par plusieurs unités de calculs simultanément, il faut y porter une attention particulière.

Le calcul parallèle sur GPU offre des bonnes performances sur le cas Toce, un petit domaine avec un petit temps de simulation. Cependant, des imprécisions apparaissent et sont de plus en plus visibles jusqu'à fausser les résultats sur des simulations plus importantes tel que le cas de test Tous.

La première piste d'étude a été le mulithreading. Le meilleur développement proposé permet un gain de 6 par rapport au temps initial avec 8 threads. Cette dernière est une hybridation d'OpenMP et de threads manuels. Le meilleur de ces deux technologies a été combiné pour atteindre un gain proche de la limite théorique établie par la loi d'Amdhal. OpenMP permet de paralléliser les calculs du pas de temps minimal, le flux et la mise à jour. Il s'agit de nombreux petits calculs sur de nombreuses données. OpenMP fournit d'excellents résultats dans ce domaine tout en étant relativement facile à prendre en main. Les threads manuels ont quant à eux permis d'exécuter l'écriture des fichiers en parallèle pendant que la boucle principale continuait son exécution. Ils offrent une gestion de bas niveau et permettent une grande flexibilité.

Le partitionnement du domaine en plus petits sous-domaines est un problème d'optimisation NP-difficile, mais intéressant. Il a fallu trouver un compromis entre un nombre équilibré de cellules par région et une frontière minimale pour chaque région. Le calcul de la distance entre deux cellules par la distance euclidienne produit de bons résultats en un temps inférieur en comparaison à l'usage de la matrice d'adjacence. Cette dernière produit un partitionnement plus regroupé au détriment d'une grande quantité de mémoire RAM consommée. Plus le nombre de régions demandé est élevé, plus il faut de temps pour trouver une division correcte. Le nombre de cellules par région s'éloigne également de la taille moyenne attendue d'une région. Heureusement, même si certaines régions sont plus grandes que d'autres, la diminution de la région la plus grande est toujours marquée lors de l'augmentation du nombre de régions.

La division en régions est une étape chronophage, mais indispensable pour l'emploi du cluster. Notre programme permet d'utiliser des partitionnements venant de différentes sources, il est tout à fait possible d'utiliser un programme plus performant pour la génération du partitionnement.

Le noeud principal du cluster attribue les différentes régions créées précédemment entre les nœuds de calculs. Par conséquent, l'emploi du cluster est justifié lorsque la simulation se fait sur une grande topologie et le temps simulé est important. La précision des calculs est similaire à l'usage d'un unique ordinateur. En outre, le gain attendu est directement lié à la taille du domaine et à la qualité de la subdivision du domaine en régions. Cependant, il faut trouver un juste compromis entre le temps nécessaire à obtenir une division de qualité additionné à la diminution du temps de calcul sur le cluster et la perte de temps liée à un mauvais partitionnement généré rapidement.

Une piste d'amélioration pour le cluster est la modification dynamique des régions. En effet, toutes les cellules qui ne sont pas remplies d'eau sont très rapides à calculer. Si une région en possède plus que ses voisines, elles pourraient se redistribuer dynamiquement les cellules frontière pour équilibrer la charge de travail.

Les trois technologies étudiées fournissent des résultats intéressants. La version la plus adaptée peut être employée en fonction des ressources matérielles disponibles et de la topologie du problème. Un sujet intéressant à approfondir pourrait être le mélange de ces différentes technologies pour en retirer le meilleur de chacune d'elle et offrir des performances décuplées.

Bibliographie

[1] M. Viñas, J. Lobeiras, B.b. Fraguela, M. Arenaz, M. Amor, J.a. García, M.j. Castro, and R. Doallo. A multi-gpu shallow-water simulation with transport of contaminants. *Concurrency and Computation : Practice and Experience*, 25(8) :1153–1169, 2012.

[2] Guido Testa, David Zuccalà, Francisco Alcrudo, Jontan Mulet, and Sandra Soares-Frazão. Flash flood flow experiment in a simplified urban district. *Journal of Hydraulic Research*, 45(sup1) :37–44, 2007.

[3] F. Alcrudo and J. Mulet. Description of the tous dam break case study (spain). *Journal of Hydraulic Research*, 45(sup1) :45–57, 2007.

[4] Shanghong Zhang, Zhongxi Xia, Rui Yuan, and Xiaoming Jiang. Parallel computation of a dam-break flow model using openmp on a multi-core computer. *Journal of hydrology*, 512 :126–133, 2014.

[5] A. Lacasta, M. Morales-Hernández, J. Murillo, and P. García-Navarro. An optimized gpu implementation of a 2d free surface simulation model on unstructured meshes. *Advances in Engineering Software*, 78 :1–15, 2014.

[6] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda : programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6. ACM, 2017.

[7] Jeffrey C Neal, Timothy J Fewtrell, Paul D Bates, and Nigel G Wright. A comparison of three parallelisation methods for 2d flood inundation models. *Environmental Modelling & Software*, 25(4) :398–411, 2010.

[8] Manuel J Castro, Sergio Ortega, Marc De la Asuncion, José M Mantas, and José M Gallardo. Gpu computing for shallow water flow simulation based on finite volume schemes. *Comptes Rendus Mécanique*, 339(2-3) :165–184, 2011.

[9] Wencong Lai and Abdul A Khan. A parallel two-dimensional discontinuous galerkin method for shallow-water flows using high-resolution unstructured meshes. *Journal of Computing in Civil Engineering*, 31(3) :04016073, 2016.

[10] Brett F Sanders, Jochen E Schubert, and Russell L Detwiler. Parbrezo : A parallel, unstructured grid, godunov-type, shallow-water code for high-resolution flood inundation modeling at the regional scale. *Advances in Water Resources*, 33(12) :1456–1467, 2010.

[11] Jesús Martínez-Frutos, Pedro J Martínez-Castejón, and David Herrero-Pérez. Fine-grained gpu implementation of assembly-free iterative solver for finite element problems. *Computers & Structures*, 157 :9–18, 2015.

[12] Shanghong Zhang, Wenda Li, Zhu Jing, Yujun Yi, and Yong Zhao. Comparison of three different parallel computation methods for a two-dimensional dam-break model. *Mathematical Problems in Engineering*, 2017, 2017.

[13] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[14] H Salehipour, GR Stuhne, and WR Peltier. A higher order discontinuous galerkin, global shallow water model : Global ocean tides and aquaplanet benchmarks. *Ocean Modelling*, 69 :93–107, 2013.

[15] Matt Bach. Estimating cpu performance using amdahls law. https://www.pugetsystems.com/labs/articles/Estimating-CPU-Performance-using-Amdahls-Law-619/. Dernière consultation : 04.04.2018.

[16] Instruments user guide : Time profiler instrument. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Instrument-TimeProfiler.html, Sep 2016. Dernière consultation : 09.12.2017.

[17] George Karypis and Vipin Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2) :278–300, 1999.

[18] Neil Immerman. Computability and complexity. https://plato.stanford.edu/entries/computability/, Jun 2004. Dernière consultation : 04.05.2018.

[19] M.e. Dyer and A.m. Frieze. On the complexity of partitioning graphs into connected subgraphs. *Discrete Applied Mathematics*, 10(2) :139–153, 1985.

[20] Gurobi. Gurobi optimizer. http://www.gurobi.com/products/gurobi-optimizer. Dernière consultation : 18.03.2018.

[21] Gurobi. An easier way to make better decisions. http://www.gurobi.com/. Dernière consultation : 18.03.2018.

[22] Christophe Geuzaine and Jean-François Remacle. Gmsh : A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11) :1309–1331, Oct 2009.

[23] Robert Sedgewick and Kevin Daniel Wayne. *Algorithms*. W. Ross MacDonald School Resource Services Library, 2017.

[24] Blaise Barney. Posix threads programming. https://computing.llnl.gov/tutorials/pthreads/. Dernière consultation : 21.03.2018.

[25] Blaise Barney. Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/. Dernière consultation : 02.03.2018.

[26] Specifications. http://www.openmp.org/specifications/. Dernière consultation : 21.03.2018.

[27] Systèmes informatiques. https://sites.uclouvain.be/SystInfo/notes/Theorie/html/Threads/threads2.html. Dernière consultation : 03.03.2018.

[28] Pthread_cond. http://manpagesfr.free.fr/man/man3/pthread_cond_init.3.html. Dernière consultation : 03.03.2018.

[29] Using the gnu compiler collection (gcc) : Optimize options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Dernière consultation : 02.03.2018.

[30] Evgueny Khartchenko and Intel. Vectorization : A key tool to improve performance on modern cpus. https://software.intel.com/en-us/articles/vectorization-a-key-tool-to-improve-performance-on-modern-cpus, Jan 2018. Dernière consultation : 23.05.2018.

[31] Wesley Bland, Dwaraka Nath, and Wes Kendall. Mpi tutorial resource. http://mpitutorial.com/. Dernière consultation : 06.05.2018.

[32] Web pages for mpi and mpe. https://www.mpich.org/static/docs/v3.2/. Dernière consultation : 25.05.2018.

[33] Intel® turbo boost technology 2.0. https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html. Dernière consultation : 05.06.2018.

[34] Reza Amouzgar. *High-performance tsunami modelling with modern GPU technology*. Newcastle University, 2017.

[35] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[36] John E Stone, David Gohara, and Guochun Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3) :66–73, 2010.

[37] Cuda c programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Dernière consultation : 03.06.2018.

[38] Unified memory for cuda beginners. `https://devblogs.nvidia.com/unified-memory-cuda-beginners/`, Apr 2018. Dernière consultation : 03.06.2018.

[39] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. An investigation of unified memory access performance in cuda. *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[40] Floating point and ieee 754 compliance for nvidia gpus. `https://docs.nvidia.com/cuda/floating-point/index.html`. Dernière consultation : 08.06.2018.

Parallelization of a hydraulic numerical simulation tool

Sandra Soares-Frazão[1], Etienne Rivière[2], Emilio Gamba, Jean-Baptiste Macq

1. Institute of Mechanics, Materials and Civil Engineering, Université catholique de Louvain, Belgium
2. Institute of Information and Communication Technologies, Electronics and Applied Mathematics, Université catholique de Louvain, Belgium

Abstract

The need for hydraulic models used for flood modeling and forecasting has risen due to the awareness of natural disasters such as dam breaks and inundations. Considering that an event such as the Tous Dam break that occurred in Spain in 1982 (Alcrudo and Mulet, 2007) resulted in a 40-hour flood, fast simulation tools are needed to reproduce and study such events in reasonable computational times. This can only be achieved using parallel computing. In this paper, three different parallelization methods are investigated and their efficiency is compared using two different test cases. These were developed during a joint European project named "Investigation of extreme flood Processes And uncertainty" (IMPACT). The first case, described by Testa et al. (2007) is a flash flood flow experiment in a scale model of the Toce river (Italy) with a simplified urban district. In this case, the flow duration is about one minute. The second test case is the Tous Dam break and the subsequent flooding of the city of Sumacárcel. Both tests were designed to be used as case studies for testing and validation of flood propagation and dam breaching mathematical models.

The SFlow2D program was developed to simulate large-scale flows on complex topologies using only one computational core. Simulating the Tous dam break using SFlow2D takes up to 54 hours to complete for a 40-hour simulation while the Toce flash flood takes approximately two minutes to complete. Parallelization is therefore needed to reduce the computational time. The parallelization of a software requires its in-depth analysis and understanding. The structure of the SFlow2D program to be parallelized is as follows : First, the program reads the input files that specify the domain. Then it alternates the computation of the minimal time step, the computation of exchanged fluxes between cells through their interfaces and the update of the cell content while the end time has not been reached. The main parallelizable steps are the flux computation and the cell update. As the program was written for complex topologies using unstructured meshes, this increases the difficulty of the parallelization of the simulation tool.

In this work, three parallelization methods were explored. The first method involves parallel computation on a single computer using all the available cores. The second method speeds up the execution time from offloading parts of the computation to the GPU whose computational power is much higher. The third and last method consists of high performance computing on clusters.

The multithreaded version aimed at using all available cores in one single computer has been developed using an OpenMP and the GCC posix-thread library. OpenMP specifies a set of directives, library routines and environment for the compiler that can be used to specify high-level parallelism. OpenMP is a relatively small and simple specification reducing development complexity which allows for easier integration of multithreading into a program.

GPU was explored as a second alternative because it has become a widely used method to accelerate scientific tools. Numerous programming languages are available, but NVIDIA's proprietary framework for GPU programming, called CUDA, shows significant performance gains. Reaching speed-ups from 20 up to 120 times as described in scientific literature can only be done by thoroughly analyzing the existing software and using adequate data structures inside the GPU

memory. Efficient CUDA implementations involve a step by step decomposition of functions into kernel which execute a set of operations common to all threads that execute the kernel.

The third method is the parallelization on clusters. It requires an additional preprocessing step consisting in subdividing the domain into smaller regions, which is called a partitioning problem. The partitioning problem tries to partition the domain into regions of equal sizes while minimizing the border size between the newly formed subdomains. This problem is similar to the graph partitioning problem which happens to be NP-hard.

The partitioning method developed in the present work starts from an initial set of centroids to grow the different regions. They are found by maximizing the minimum distance between them. Each centroid then takes a (free) cell adjacent to its growing region. The program shifts to the second phase once there are no more cells to take.

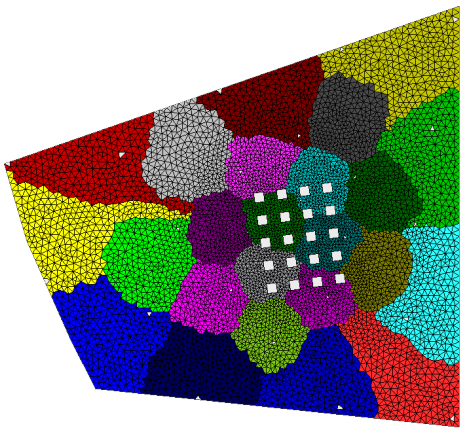


FIGURE 1 – Partitioning of Toce test case into 24 regions

The cell count of each region is computed and compared to the mean number of cells that should be present inside each region. If there are more cells than the average, the region will find a neighbor that has less cells and give away the cell that is furthest away from its centroid and closest to the centroid of the receiving region. For the opposite case, if the region has less cells than the mean it will find the neighbor that has less than a lower bound on the number of cells in a region. The region chooses the cell that is least likely to be taken back by the losing region. The exchange stops when a good partitioning is obtained, according to a given criterion. Figure 1 shows the partitioning of the Toce mesh into 24 regions using the adjacency distance between cells.

However, dividing the domain into smaller regions is not the only challenge posed by using clusters. The main challenge is to efficiently and correctly communicate data between regions and to regroup the data to produce results such as snapshots of the flow at certain timesteps. The subdivided regions all possess cells that are adjacent to their region. These cells are called ghost cells. By computing the flux exchanged with the ghost cells, sending the data to the neighboring region and then computing the update of the cell content in all regions, we were able to synchronize the data inside the whole domain.

The results of the parallelization are promising. The multithreaded version has achieved a speed-up 6,01 compared to the initial run of the SFlow2D program. Using 48 nodes of the CÉCI cluster produced a speed-up of 17,07 given that it was run on the Tous dam break test. There is a linear correlation between the speed-up and the number of nodes if the computational domain is large enough, which is the case here. Finally, the GPU implementation led to a speed up of 4,8 but this version doesn't guarantee correct results due to the introduction of imprecisions in floating point computations which in turn create incorrect results.

Further developments of the parallelized SFlow2D could integrate a second order reconstruction scheme to increase the acuracy of the results. Since each of the node's regions are built based on the input partitioning to the MPI parallelized program, it will be able to scale up with the availability of more nodes inside a cluster. As regards the GPU implementation, further research should be done to integrate multi-GPU and texture memory to better handle the irregular distribution of the computational cells.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**