UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MIPTIS
MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIEURIE DES SYSTÈMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE

présentée par :

Hélène COULLON

soutenue le : **XX mois 2014**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

TITRE

THÈSE DIRIGÉE PAR :

Sébastien LIMET Professeur des Universités, Université d'Orléans

RAPPORTEURS :

Prénom NOM Titre, établissement
Prénom NOM Titre, établissement

JURY :

Prénom NOM Titre, établissement, Président du jury
Prénom NOM Titre, établissement
Prénom NOM Titre, établissement
Prénom NOM Titre, établissement

À mes parents
"Quand je serai grande, je voudrais inventer des choses"

Table des matières

Liste des figures

Introduction 1

Sommaire

1.1 Contexte de la recherche

Les calculs scientifiques, notamment dans le domaine de la simulation numérique, ont toujours été consommateurs de ressources informatiques. Dans les années 60, à l'apparition des premiers super-calculateurs, les calculs les plus demandeurs en ressources étaient déjà les calculs scientifiques. Depuis, ce besoin de puissance de calcul n'a fait qu'augmenter, donnant naissance à des architectures parallèles toujours plus complexes à utiliser. La demande de puissance de calcul, ou de performance dans les architectures parallèles, est a priori sans limites pour plusieurs raisons. Tout d'abord, les données utilisées par les scientifiques sont de plus en plus précises et donc de plus en plus lourdes et longues à traiter. Cette précision des données vient, par exemple, de la progression des techniques d'acquisition, comme pour les données géo-référencées obtenues par des technologies à laser (la télédétection LIDAR, par exemple). Cette précision peut également être obtenue en complexifiant le maillage associé au domaine d'étude dans les simulations numériques. En effet, dans les simulations, le domaine d'étude est généralement discrétisé en un maillage afin de pouvoir être traité numériquement. Plus ce maillage est précis, plus la simulation est précise et plus les calculs à effectuer sont nombreux, et il est en pratique possible d'augmenter très largement la précision du maillage (sous réserve des capacités de précision du matériel). De plus, il est évident qu'il est également possible d'augmenter la taille du domaine d'étude de façon importante, ce qui est également à l'origine des besoins croissants de puissance et de parallélisme. Enfin, si les maillages des simulations numériques peuvent être complexifiés, les méthodes numériques et les calculs peuvent l'être également. Un exemple très parlant de demande croissante de puissance de calcul est le traitement des modèles météorologiques. Il est potentiellement toujours possible d'ajouter des facteurs aux modèles, et de les complexifier, mais également d'augmenter la taille du domaine étudié etc. Toutes ces modifications rendent les calculs plus longs, demandant plus de puissance et plus de parallélisation, mais permettent d'obtenir des prévisions plus précises.

Il résulte donc de cette demande croissante de puissance, la création d'architectures parallèles, c'est-à-dire des architectures qui intègrent plusieurs processeurs, voire plusieurs machines. De cette manière, il est possible d'obtenir plus de puissance de calcul sans attendre les prochaines générations de processeurs. L'évolution de ces machines parallèles nous a mené à des architectures matérielles de plus en plus compliquées et parfois hétérogènes, mélangeant alors plusieurs machines (*cluster* ou *cloud computing*), plusieurs cœurs et plusieurs processeurs mais aussi des calculateurs graphiques (*GPU*). Les scientifiques se retrouvent ainsi à devoir utiliser des architectures parallèles très complexes pour pouvoir proposer des résultats pertinents et intéressants pour la communauté, sans en maîtriser l'utilisation, comme d'ailleurs la plupart des informaticiens également. Le calcul parallèle est donc devenu rapidement un domaine d'expertise que peu de personnes maîtrise. Il n'est par conséquent pas envisageable que chaque scientifique non-informaticien apprenne à programmer sur ces architectures et devienne un expert du parallélisme, par manque de temps, d'argent, et de personnes pouvant les former. De plus, si une formation de base est quant à elle envisageable, elle ne sera pas suffisante pour exploiter la pleine

puissance de ces machines. Même si cela se pratique dans certains cas, il paraît également délicat d'imaginer des collaborations entre des scientifiques non-informaticiens et des experts du parallélisme pour chaque code parallèle nécessaire. Pour ces raisons sont nés, quasiment avec l'apparition des architectures parallèles, des outils et langages facilitant leur programmation. Des modèles de programmation ont tout d'abord été proposés, puis des langages et des bibliothèques parallèles, mais la complexité grandissante des architectures a également fait évoluer ces solutions vers des solutions de parallélisme implicite qui cachent partiellement ou totalement le parallélisme à l'utilisateur. Nous parlons alors de parallélisme implicite partiel, lorsque les outils proposés cachent partiellement les technicités du parallélisme, ou total, lorsque les outils cachent intégralement le code parallèle à l'utilisateur. Nous classons les solutions de parallélisme implicite total suivant trois grands types : les bibliothèques générales ; les solutions à patrons ; et les langages et bibliothèques spécifiques. Dans le premier type une grande flexibilité est proposée et les solutions permettent de s'adresser à tout type de traitements. Le second type propose, quant à lui, un niveau d'abstraction très intéressant, et permet également de structurer le code et de donner des repères simples à l'utilisateur. Enfin, le troisième type de parallélisme implicite total est spécifique à un problème précis et propose des optimisations et une efficacité pour ce problème. Le langage ou la bibliothèque est également plus simple d'utilisation car proche du problème spécifique. Il s'agit des solutions les plus efficaces mais également les moins flexibles.

1.2 Contributions

Cette thèse se place dans le cadre du parallélisme implicite pour les simulations numériques. Nous proposons, tout d'abord, dans ce travail de thèse un modèle de programmation parallèle implicite, nommé SIPSim pour *Structured Implicit Parallelism for scientific Simulations*. Ce modèle présente une approche systématique pour proposer des solutions de parallélisme implicite pour les simulations numériques basées sur des maillages. Le modèle est basé sur quatre composants permettant de cacher intégralement le parallélisme à l'utilisateur et d'obtenir un style de programmation proche du séquentiel. Le modèle SIPSim se positionne à l'intersection des travaux existants en tentant de conserver les avantages de chacun. Ainsi, le modèle a la particularité d'être à la fois efficace, car spécifique au cas des simulations numériques à maillages, d'un niveau d'abstraction permettant de conserver une programmation proche du séquentiel, ce qui garantit un effort de programmation faible, tout en restant très flexible et adaptable à tout type de simulations numériques.

Afin de valider le modèle SIPSim, une implémentation est proposée dans cette thèse, sous le nom de SkelGIS. SkelGIS est une bibliothèque C++ constituée uniquement de fichiers d'en-tête, ou autrement appelée *header-only*, implémentée en suivant le modèle SIPSim pour deux cas de maillages différents : les maillages cartésiens à deux dimensions, et les compositions de maillages issues des simulations sur des réseaux. Dans le premier cas, de nombreuses solutions de parallélisme implicite existent, toutefois, en suivant le

modèle SIPSim, SkelGIS se place à l'intersection de plusieurs solutions et mêle à la fois l'efficacité et la flexibilité de façon intéressante. Le deuxième cas, quant à lui, initie un travail sur des simulations d'un genre plus complexe, où une composition de maillages est effectuée. La flexibilité du modèle SIPSim est alors mise en avant, et ce type de travaux n'a, à notre connaissance, jamais été proposé en parallélisme implicite. La bibliothèque SkelGIS est évaluée en termes de performance et d'effort de programmation sur deux cas réels d'application, développés et utilisés par des équipes de recherche en mathématiques appliquées. Ces évaluations permettent donc, tout d'abord, de valider que SkelGIS (et le modèle SIPSim) répond aux besoins de simulations complexes, et donc aux besoins des scientifiques. Les performances obtenues sont comparées à des implémentations MPI et OpenMP des mêmes simulations. Sur l'ensemble de ces évaluations, et pour un effort de programmation beaucoup moins important, les performances obtenues sont très compétitives et proposent de meilleurs temps d'exécution. Nous montrons ainsi que SkelGIS propose des solutions efficaces et flexibles, tout en cachant intégralement le parallélisme à l'utilisateur et tout en conservant un style de programmation séquentiel.

La plupart des travaux présentés dans cette thèse ont fait l'objet de publications.

Publications dans des conférences internationales

1. Hélène Coullon, Sébastien Limet. *Implementation and Performance Analysis of SkelGIS for Network Mesh-based Simulations.* Euro-Par 2014.
2. Hélène Coullon, Jose-Maria Fullana, Pierre-Yves Lagrée, Sébastien Limet, Xiaofei Wang. *Blood Flow Arterial Network Simulation with the Implicit Parallelism Library SkelGIS.* ICCS 2014.
3. Hélène Coullon, Sébastien Limet. *Algorithmic skeleton library for scientific simulations : Skelgis.* In HPCS, pages 429-436. IEEE, 2013.
4. Hélène Coullon, Minh-Hoang Le, Sébastien Limet. *Parallelization of shallow-water equations with the algorithmic skeleton library skelgis.* In ICCS, volume 18 of Procedia Computer Science, pages 591–600. Elsevier, 2013.

Publications dans des journaux internationaux

1. Stéphane Cordier, Hélène Coullon, Olivier Delestre, Christian Laguerre, Minh-Hoang Le, Daniel Pierre, and Georges Sadaka. *Fullswof paral : Comparison of two parallelization strategies (mpi and skelgis) on a software designed for hydrology applications.* ESAIM : Proceedings, 43 :59–79, 2013.

Publications en cours d'évaluation dans des journaux internationaux

1. Hélène Coullon, Minh-Hoang Le, Sébastien Limet. *Implicit parallelism applied to shallow-water equations using SkelGIS.* In *Concurrency and Computation : Practice and Experience.*

1.3 Organisation du manuscrit

Ce manuscrit est organisé en cinq chapitres supplémentaires dont voici le contenu :

— Nous étudions dans le chapitre 2 l'état de l'art nécessaire à la bonne compréhension de ce manuscrit. Cet état de l'art est composé d'un historique et d'une présentation des architectures parallèles, et des modèles et outils de parallélisation qui y sont associés. Nous donnons ensuite une introduction sur les simulations numériques basées sur des maillages, ce qui permet de rendre plus clairs les cas d'application de cette thèse. Les bases nécessaires pour la compréhension des problèmes de décompositions de domaines et de partitionnements de graphes, évoqués dans cette thèse, sont ensuite abordés. Une description détaillée des solutions de parallélisme implicites disponibles dans la littérature est ensuite proposée et permettra le positionnement de notre travail dans ce contexte. Enfin, nous présentons les mesures de performance et d'effort de programmation utilisées dans ce manuscrit.

— Le chapitre 3 présente le modèle de programmation implicite SIPSim. Il détaille pour cela les quatre composants qui le constitue et le type de programmation obtenu en adoptant ce modèle.

— Dans le chapitre 4 est ensuite détaillé la première implémentation du modèle SIPSim, SkelGIS, pour le cas des maillages à deux dimensions cartésiens. Deux cas d'application réels sont également détaillés et évalués dans ce chapitre.

— Le chapitre 5 continue la description de l'implémentation SkelGIS dans le cas des simulations sur les réseaux, afin de valider davantage le modèle SIPSim. Une section particulière précise l'implémentation et l'optimisation de la structure de données distribuée, puis une section supplémentaire offre de nouveau un cas d'application réel afin d'évaluer la solution. Pour terminer ce chapitre, le problème de partitionnement des réseaux est évoqué et deux solutions plus récentes sont présentées.

— Enfin, dans le chapitre 6 est présenté un bilan des travaux présentés dans cette thèse, mais également un ensemble de perspectives de recherche.

Etat de l'art

2

Sommaire

Dans ce chapitre vont être introduites les notions nécessaires à la bonne compréhension de ce manuscrit. Afin, tout d'abord, de comprendre les problématiques engendrées par l'utilisation des machines parallèles, nous étudierons rapidement les évolutions des architectures parallèles, et les modèles algorithmiques et de programmation qui en découlent. Par la suite, nous étudierons des notions sur le calcul et la simulation scientifique, qui représentent les domaines d'application de cette thèse. Deux états de l'art précis seront ensuite nécessaires, sur la décomposition de domaine, et le parallélisme implicite. Ces notions représentent, en effet, les problématiques informatiques principales de cette thèse. Enfin, cette thèse présentera un certain nombre de résultats expérimentaux, nous terminerons donc ce chapitre par une discussion des choix effectués pour l'évaluation des performances et de la difficulté de programmation des solutions.

2.1 Calcul parallèle : architectures et programmation

Dans cette première section de l'état de l'art, nous allons introduire les notions de base du parallélisme, nécessaires à la compréhension de cette thèse. Nous commencerons par un historique rapide des architectures parallèles, puis une description des architectures actuelles. Nous décrirons ensuite les principaux modèles de parallélisation disponibles pour programmer ces architectures.

2.1.1 Architectures parallèles

C'est en 1965 qu'a été exprimée dans "Electronics Magazine" la première loi de Moore, qui n'était alors qu'un postulat fondé sur une simple observation. En effet, Gordon Moore constata que la complexité des semiconducteurs doublait tous les ans, depuis leur apparition en 1959. Le postulat consistait alors à supposer la poursuite de cette croissance. C'est également dans les années 60 que sont apparus les premiers super-calculateurs, dont le but initial était d'effectuer une exécution la plus rapide possible des instructions d'un programme. Toutefois, bien que le monde informatique se basait alors sur la loi de Moore, et donc sur la montée en puissance des semi-conducteurs (puis plus tard sur la miniaturisation des transistors), l'idée de machine parallèle est apparue très rapidement. En effet, de par la demande grandissante de performances, notamment pour les calculs scientifiques, il est vite devenu difficile de devoir attendre la sortie d'une nouvelle gamme de processeurs pour obtenir plus de puissance de calcul. Il n'était, de plus, pas envisageable de racheter l'ensemble du matériel régulièrement. Ainsi, l'idée de faire collaborer ensemble plusieurs ordinateurs pour obtenir un résultat plus rapidement se concrétisa dans les années 60. On parla alors, pour la première fois, d'architectures multiprocesseurs, un terme qui désignait alors une architecture SMP (*Symetric Multi Processor*). Encore utilisée à l'heure actuelle, sous une forme plus moderne, l'architecture SMP représente un ensemble de processeurs identiques dans une même machine, qui partagent une mémoire vive commune.

Dans les années 1970, l'architecture des super-calculateurs évolua, proposant l'utilisation de processeurs vectoriels. Ces processeurs furent les plus puissants de leur génération

et connurent un grand succès. Leur puissance était due à leur capacité à appliquer une même instruction à des parties différentes des données, de façon simultanée. Plusieurs processeurs vectoriels ont ensuite été utilisés en parallèle pour obtenir toujours plus de puissance.

Ce n'est qu'en 1975, et suite à l'apparition des transistors, que Gordon Moore réévalua sa première loi sous la forme d'une deuxième loi qui supposait que le nombre de transistors sur une puce pouvait doubler tous les deux ans. Une mauvaise interprétation de cette loi fût longtemps énoncée. Un amalgame y était effectué entre le nombre de transistors sur une puce et la fréquence d'horloge d'un processeur. Cette loi erronée s'est pourtant avérée exacte jusqu'au début des années 2000, avant de poser des difficultés de dissipation thermique pour des fréquences trop importantes. Dans les années 80, suite à la miniaturisation des transistors, sont apparus les microprocesseurs. Leur puissance était modeste, mais leur faible encombrement et leur faible consommation ont permis l'apparition, puis la démocratisation, des ordinateurs personnels. Le perfectionnement des techniques de miniaturisation a permis une croissance importante de la puissance des microprocesseurs. Les microprocesseurs sont ainsi devenus très rapidement compétitifs, en terme de performances, face aux processeurs vectoriels. Leur faible coût de fabrication en ont fait les processeurs les plus utilisés dans les architectures parallèles. Nous allons décrire avec plus de précision, dans la suite de cette section, les différentes architectures parallèles issues de l'apparition des microprocesseurs.

2.1.1.1 Architectures à mémoire partagée

Architectures SMP. Comme nous l'avons décrit précédemment, cette architecture parallèle (la plus ancienne), consiste à regrouper plusieurs processeurs au sein d'une même machine et de leur faire partager une même mémoire vive. Cette architecture a naturellement été étendue aux microprocesseurs, toutefois, il n'est pas possible d'utiliser cette architecture parallèle en augmentant à l'infini le nombre de processeurs. En effet, les processeurs d'une architecture SMP entrent tous en concurrence pour lire et écrire dans la mémoire commune, ce qui implique que l'ajout de processeurs à cette architecture ne peut être efficace que si la mémoire est capable d'alimenter les processeurs supplémentaires en données à traiter. Les microprocesseurs ont très rapidement été plus rapides que les mémoires, créant une limitation physique à cette architecture.

Architectures NUMA. Les architectures à mémoire non uniforme, NUMA (*Non Uniform Memory Access*), définissent pour chaque processeur (ou petit groupe de processeurs) l'attribution d'une sous-partie de mémoire en accès direct et très rapide. Chaque processeur ne pourra accéder aux données des autres mémoires qu'à travers un bus d'interconnexion, plus lent. Cette architecture vise à améliorer l'architecture SMP en réduisant le goulot d'étranglement dû à la mémoire commune à l'ensemble des processeurs.

Architectures multi-cœurs et many-cœurs. Dans le but d'augmenter la puissance des microprocesseurs, sans en augmenter la fréquence, le concept de cœurs multiples

(*multicore*) est apparu en 2001. Cette architecture peut être vue comme une unique puce regroupant plusieurs microprocesseurs. Dans cette architecture, la proximité des cœurs de calcul permet une communication plus rapide entre les différentes mémoires des différents cœurs. Certaines architectures proposent même un système de mémoire cache partagée par les cœurs. Le principe de la mémoire cache, ou mémoire tampon, est de fournir dans les architectures modernes une mémoire très proche des processeurs (ou cœurs) et permettant un accès plus rapide aux données. Cette mémoire, comme son nom l'indique sert de tampon entre la mémoire vive et les unités de calcul, elle réduit donc les délais d'accès aux données. La mémoire cache est composée de plusieurs niveaux. Le niveau L1, ou cache interne, est le plus rapide et le plus proche des unités de calcul. La mémoire cache fonctionne par chargement de lignes de cache. Une ligne de cache est de taille limitée et dépend du matériel présent dans le processeur. Lorsqu'un processeur a besoin d'accéder à une donnée pour une opération arithmétique, cette donnée, et ses données contiguës en mémoire, sont chargées dans une ligne de cache (suivant sa limite de taille). Si une donnée non présente dans le cache est nécessaire (on parle alors de défaut de cache), une ligne de cache est invalidée et une nouvelle ligne devra à son tour être chargée etc. Ces nouveaux processeurs ont permis, tout comme l'architecture NUMA, de réduire le goulot d'étranglement des architectures SMP. De plus ces architectures ont permis de démocratiser les architectures parallèles dans les ordinateurs personnels. Plus récemment sont apparus les architectures multiprocesseurs *Intel MIC* pour *Intel Many Integrated Core Architecture*, dont, par exemple, le très médiatique accélérateur Xeon Phi. Ces architectures regroupent plusieurs processeurs possédant chacun un très grand nombre de cœurs. On parle alors d'architectures many-cœurs. Le nombre total de cœurs est très important dans ces architectures et donne accès au calcul massivement parallèle (tout comme les accélérateurs graphiques dont nous parlerons par la suite), sans pour autant devoir apprendre de nouvelles interfaces de programmation (contrairement aux accélérateurs graphiques).

2.1.1.2 Architectures à mémoire distribuée

Architecture en grappe. L'architecture en grappe, aussi appelée un *cluster* en anglais, consiste à connecter entre elles, par un réseau d'interconnexion à très haut débit, plusieurs machines (ou nœuds) indépendantes matériellement homogènes. Dans une architecture en grappe, et contrairement à une architecture à *mémoire partagée*, chaque nœud est indépendant et possède donc sa propre mémoire à laquelle les autres nœuds n'ont pas accès. Les nœuds sont donc amenés, dans ce type d'architectures, à communiquer au travers du réseau d'interconnexion. Un réseau à très haute performance étant très coûteux, les machines d'une grappe sont géographiquement les plus proches possible, dans une même pièce ou dans une même armoire de rangement. Il s'agit d'une approche simple mais très favorable au calcul haute performance pour plusieurs raisons. Tout d'abord, outre la rapidité du réseau, un travail sur la topologie des réseaux peut rendre plus rapides les communications entre certains nœudss du cluster. Une topologie proche de la topologie des données utilisées peut donc, par exemple, favoriser les performances d'un parallélisme de données. De plus, une grappe est composée d'un grand

nombre de processeurs identiques ce qui permet de favoriser l'optimisation d'un type de matériel donné. Avec la démocratisation des architectures multi-processeurs et multi-cœurs, les grappes sont aujourd'hui composées d'un certain nombre de nœuds ou d'unités de calcul indépendantes à mémoire partagée. En théorie, ces architectures peuvent donc être considérées comme des architectures à mémoire distribuée, mais également comme des architectures à mémoire hybride.

Architecture en grille. Le concept de grille est un concept proche du concept de grappe. Toutefois, une grille consiste à relier entre elles des ressources de calcul hétérogènes (ordinateurs, grappes, serveurs etc.) et potentiellement distantes. En effet, le but d'une grille est d'utiliser la puissance de calcul disponible à plusieurs endroits pour proposer une unique architecture virtuelle possédant des ressources de calcul très importantes et extensibles. L'utilisateur d'une grille ne fait que soumettre le lancement de calculs et n'aura aucune information sur les machines utilisées pour son calcul. La gestion de ce type d'architecture est donc très complexe et son utilisation est souvent restreinte aux calculs massivement parallèles (*embarrassingly parallel*), qui consistent à effectuer le même traitement un grand nombre de fois, ce qui ne provoque aucune communication entre les ressources. Pour des calculs parallèles nécessitant des communications entre les ressources de calcul, une unique ressource de la grille est généralement utilisée, on retrouve alors la notion de serveur ou de cluster, par exemple. Les ressources d'une grille pouvant être géographiquement éloignées, le réseau d'interconnexion reliant les ressources n'est généralement pas un réseau à très haut débit, trop coûteux. Toutefois, la plateforme expérimentale Grid'5000 relie, par exemple, une dizaine de grappes de plusieurs villes françaises au travers du réseau à haut débit RENATER (Réseau national de télécommunications pour la technologie, l'enseignement et la recherche).

Cloud computing. Le *nuage*, plus communément appelé le *cloud*, est un service mettant à disposition des ressources de calcul ou de stockage distantes. Le cloud ressemble donc aux concepts de grille et de cluster. Toutefois, il se distingue par plusieurs points. Tout d'abord, le concept de cloud est un service grand public qui s'ouvre à tous et qui est généralement payant. Les clusters, comme les grilles, sont souvent des outils réservés à des utilisateurs précis sur une période de temps limitée. L'accès y est gratuit mais une demande doit être mise en place pour utiliser ce type d'architectures. De plus, le cloud, contrairement à la grille, n'a pas été pensé pour l'accès à des ressources délocalisées. Très souvent les ressources d'un cloud appartiennent à une unique entité et sont regroupées géographiquement dans un endroit (bien que des travaux regroupant plusieurs cloud existent également). De même, le cloud se distingue de la grille par le fait qu'il n'est pas été mis en place dans l'idée de relier des architectures hétérogènes. Le cloud est donc à la frontière des clusters et des grilles, mais dans une optique de service commercial et grand public.

2.1.1.3 Architectures hétérogènes

Retour de la vectorisation. Toujours afin d'augmenter la puissance des microprocesseurs, sans en augmenter la fréquence, les capacités de vectorisation ont été réintroduites dans les microprocesseurs scalaires modernes. On peut notamment évoquer les instructions SSE (*Streaming SIMD Extension*), qui sont associées à des registres de 128 bits sur lesquels il est possible d'effectuer quatre opérations simultanées sur des nombres flottants de 32 bits, ou deux opérations simultannées sur des nombres flottants de 64 bits etc. La version la plus récente SSE4 donne accés à 47 types d'instructions. Les jeux d'instructions AVX2 (*Advanced Vector Extensions 2*) et AVX-512, plus récents, proposent des opérations simultanées sur des registres de 256 et 512 bits, ce qui augmente encore les possibilités de calcul des microprocesseurs. Les registres vectoriels, ajoutés aux microprocesseurs modernes, offrent une hétérogénéité d'architecture proposant des performances très intéressantes.

Accélérateurs graphiques. Les processeurs graphiques, ou GPU, sont initialement apparus pour effectuer des calculs performants et spécifiques à l'affichage graphique, comme par exemple le rendu d'images en trois dimensions. Ils ont rapidement été massivement parallélisés, de par la nature répétitive de leurs calculs. Initialement, ces processeurs graphiques étaient cantonnés à un certain nombre de fonctionnalités, puis ils sont devenus programmables, ce qui a initié une déviation de leur utilité première, pour des calculs autres que graphiques. On ne parle alors plus de GPU mais de GPGPU. Un GPGPU propose des unités de calcul alternatives aux CPU, et massivement parallèles. Toutefois, il est important de noter qu'un GPU ne peut complètement s'abstraire d'un CPU pour fonctionner, ce dernier permettant de charger des programmes et des données en mémoire vive. Il s'agit donc d'une architecture parallèle hétérogène. Les GPU étant peu coûteux et consommant peu d'énergie, leur utilisation dans les grands centres de calcul internationaux devient fréquente. Dans ce cas, les GPGPU sont présents sur chaque nœud du cluster et servent à effectuer les calculs. Les CPU, quant à eux servent à charger en mémoire les programmes et les données et également à gérer les communications sur le réseau d'interconnexion.

Architectures hybrides. Une architecture hétérogène, ou hybride apparaît comme évidente après la description des architectures précédentes. Il s'agit de réutiliser l'approche à mémoire partagée au sein de l'approche à mémoire distribuée. Cette architecture permet d'augmenter le nombre total de processeurs utilisés et de répartir les faiblesses sur plusieurs goulots d'étranglement au lieu d'un seul. Ce type d'architectures est devenu une référence et est très utilisé parmi les grappes les plus puissantes du monde. Les architectures hétérogènes peuvent être de type grappe/NUMA, grappe/multi-cœurs (ou plus généralement grappe/CPU), mais également grappe/GPU etc.

2.1.2 Paradigmes et modèles de parallélisation

Avec l'apparition des architectures parallèles sont apparus les premiers problèmes de programmation parallèle. En effet, un programme séquentiel en lui même, et sans l'aide particulière du système d'exploitation ou de tout autre système externe de répartition de charge sur les cœurs ou les processeurs, n'exploite pas directement les ressources d'une machine parallèle. Or, la conception d'un programme parallèle peut s'avérer très complexe et très dépendante des architectures utilisées. Avec l'apparition des machines parallèles sont donc apparus également des paradigmes de programmation parallèle, offrant un ensemble d'approches générales pour envisager un programme parallèle, puis des modèles de programmation parallèle, permettant de concevoir de façon plus précise des programmes sur ces machines. Les paradigmes de programmation parallèle représentent donc un niveau d'abstraction plus bas et moins précis que les modèles de programmation parallèle. Les modèles de programmation parallèle, même si certains sont naturellement induits par le matériel, peuvent être implémentés pour différentes architectures parallèles. On distingue donc les modèles de programmation des implémentations qui y sont associées pour un modèle d'exécution donné. Par exemple un modèle de programmation parallèle initialement pensé pour des architectures à mémoire distribuée pourrait être implémenté sur une architecture *DSM* (*Distributed Shared Memory*) [79] qui permet de construire un espace mémoire partagé pour tous les processeurs, bien que cet espace mémoire soit physiquement distribué. Nous décrivons dans cette partie quelques uns des paradigmes et des modèles de programmation parallèle les plus utilisés et les plus connus. Nous ne décrirons en revanche pas leurs implémentations pour différents modèles d'exécution.

2.1.2.1 Paradigmes de programmation parallèle

Taxinomie de Flynn. En 1972, Michael J. Flynn définit une classification des architectures des ordinateurs [54]. Quatre classes étaient alors répertoriées et sont représentées dans la table 2.1. La première classe, nommée *Single Instruction, Single Data* (SISD) représente les machines séquentielles n'exploitant aucun parallélisme. La deuxième classe, *Singe Instruction, Multiple Data* (SIMD), représente les machines pouvant appliquer une unique instruction à un ensemble de données. Cette classe concerne donc typiquement les architectures vectorielles ou GPU. La troisième classe, *Multiple Instruction, Single Data* (MISD), représente les machines permettant d'appliquer plusieurs instructions à la suite sur un même donnée d'entrée. Cette classe concerne typiquement les programmes de type *pipeline* ou les systèmes de tolérance aux pannes cherchant à comparer deux résultats issus d'une même donnée. Enfin, la quatrième classe, *Multiple Instruction, Multiple Data* (MIMD), représente les machines multiprocesseurs qui peuvent exécuter simultanément des instructions différentes sur des données différentes.

Cette classification est toujours utilisée dans le parallélisme actuel, mais sous une autre forme. En effet, les différentes classes ne sont plus représentatives d'architectures matérielles particulières, la plupart des machines répondant à l'ensemble de ces classes. En revanche, les classes de Flynn représentent désormais des paradigmes de programmation

	instruction unique	instructions multiples
donnée unique	SISD	MISD
données multiples	SIMD (SPMD)	MIMD (MPMD)

TABLE 2.1 – *Taxinomie de Flynn*

parallèle, très souvent associés aux paradigmes de parallélisation de tâches et de données, que nous allons décrire.

Parallélisme de tâches. Ce paradigme de programmation cherche à diviser un programme en un ensemble de tâches qui peuvent être dépendantes, mais aussi indépendantes. Dans ce cas c'est l'exécution du programme qui cherche à être parallélisée. Les paradigmes de parallélisation MISD et MIMD peuvent être associés au parallélisme de tâches. Dans le premier cas, des opérations successives sont appliquées sur un jeu de données d'entrée, on appelle communément ce type de calcul un *pipeline*. L'instruction $i+1$ ne peut alors être exécutée qu'une fois l'instruction i terminée. Toutefois, sur des données d'entrées suffisamment nombreuses, le parallélisme peut apparaître en quinconce. En effet étant donné une donnée d'entrée $[x_1, \ldots, x_n]$, une fois x_1 calculé pour l'instruction i, il est possible de calculer simultanément x_2 pour l'instruction i et x_1 pour l'instruction $i+1$. Le paradigme MIMD, quant à lui, est rencontré plus fréquemment et offre plus de possibilités de parallélisation. Dans ce cas, on cherchera à identifier des tâches travaillant sur des données différentes, ce qui rend les tâches indépendantes les unes des autres. Toutefois, le développeur devra se charger de synchroniser les différentes tâches ensemble afin de garantir la cohérence des résultats. Nous pouvons enfin noter la paradigme MPMD (*Multiple Program, Multiple Data*), qui étend le concept MIMD à des programmes. Ainsi, chaque processeur peut appliquer un ou plusieurs programmes qui lui sont propres à des données éventuellement différentes des autres processeurs de façon indépendante. Les synchronisations nécessaires au bon fonctionnement du programme parallèle sont alors à la charge de l'utilisateur.

Parallélisme de données. Dans ce paradigme, le parallélisme se focalise sur la façon dont les données sont distribuées sur les différents processeurs. L'ensemble des processeurs effectuent alors le même jeu d'instructions sur des données d'entrée qui leurs sont propres. Dans ce type de parallélisme les tâches effectuées par le programme sont peu modifiées. Il faut toutefois réflechir et consevoir les communications, les échanges ou les synchronisations nécessaires entre les processeurs pour que le programme parallèle soit correct et donne le même résultat qu'en séquentiel. Les paradigmes SIMD et SPMD (*Simple Program, Multiple Data*) sont associés au parallélisme de données. Ils représentent le même type de parallélisation, toutefois SIMD est associé aux architectures vectorielles et GPU, où la notion d'instruction est clairement définie et synchrone. L'approche SPMD est plus vaste et moins tournée vers la solution matérielle. Elle peut s'appliquer à des architectures à mémoire partagée comme distribuée. Ce paradigme considère une exécution indépendante d'un programme sur chaque processeur, et sur des données différentes,

et met à la charge du programmeur les synchronisations nécessaires à la cohérence du calcul général. Ce type de parallélisation est l'une des plus utilisée, notamment pour les architectures à mémoire distribuée.

Paradigmes induits par le matériel. Nous abordons maintenant deux paradigmes de programmation parallèle connus et induits par le matériel, qui sont utilisés par la plupart des modèles présentés par la suite. Dans les architectures à mémoire partagée, les processus peuvent interagir par l'écriture et la lecture dans des espaces mémoire partagés et communs. Ces architectures permettent donc des interactions entre les processus par la simple utilisation de la mémoire de la machine, mais font intervenir des problèmes de concurrence d'accès aux données ainsi que de cohérence ou d'intégrité des données. Le paradigme de programmation parallèle le plus utilisé pour gérer ces problématiques, et que nous appelons *paradigme à verrous*, consiste à fournir des mécanismes permettant d'assurer l'exclusion temporaire de l'accès aux données pour en garantir l'intégrité. Le mécanisme le plus couramment utilisé se base sur des verrous d'exclusion mutuelle, appelés *mutex*. Un verrou sur une variable n'est attribué qu'à un unique processus, ce qui garantit qu'aucun autre processus ne pourra accéder ou écrire dans cette variable jusqu'à l'obtention, à son tour, d'un verrou. Pour ce qui est des architectures à mémoire distribuée, le paradigme le plus naturel, et parmi les plus utilisés, de *passage de messages*, est venu du simple constat que, dans ces architectures, les processus ne partagent pas d'espace d'adressage commun et qu'il est nécessaire d'échanger des messages pour que ces processus puissent communiquer entre eux. Ce paradigme n'introduit donc pas de problèmes de concurrence et d'intégrité des données, mais un problème de communication. Le niveau d'abstraction le plus bas pour mettre en place ce paradigme consiste à utiliser le réseau des machines et donc à faire, par exemple, appel à la programmation de *sockets Unix*, qui permettent l'envoi d'octets à destinations d'une adresse réseau spécifique. De nombreux modèles de programmation parallèle sont issus de ce paradigme.

2.1.2.2 Modèles de programmation parallèle

Mémoire partagée. Les modèles de programmation induits des architectures à mémoire partagée sont très souvent basés sur du parallélisme de tâches, mais peuvent également se baser sur du parallélisme de données. Le standard des threads POSIX (ou pthreads) [96] est l'un des modèles les plus répandus du parallélisme pour architectures à mémoire partagée. Ce modèle est basé sur le paradigme de verrous évoqués dans la partie précédente. Il permet de définir la création d'un nouveau processus léger (appelé un *thread*), dont l'exécution sera gérée par le système, en suivant une politique d'ordonnancement. A sa création, une tâche est assignée au thread et sera effectuée en parallèle du programme principal, qui pourra continuer son exécution. Un certain nombre de routines permettent ensuite des synchronisations entre les processus créés, et permettent de poser des verrous pour la modification de données.

Le modèle de directives OpenMP [34], qui sera décrit avec précision dans la suite de cette thèse, est le deuxième modèle très utilisé sur les architectures à mémoire partagée. Il permet d'ajouter du *multi-threading* (le fait de créer plusieurs processus légers

pour certaines tâches du programme) dans du code C, C++ ou Fortran, par l'ajout de directives, sans modifications majeures du code, mais avec des résultats de performance limités. Ce modèle est principalement basé sur la parallélisation de boucles, ou sur le parallélisme de tâche dans lequel il est explicitement indiqué quels sont les différents travaux disponibles pour les threads. Il est également demandé à l'utilisateur de déclarer les variables locales et partagées du programme, afin de positionner automatiquement, par la suite, des exclusions mutuelles pour l'accès aux données partagées.

Enfin, notons qu'il existe un modèle de programmation parallèle, nommé PGAS [6] (Partitioined Global Adress Space), basé sur le concept d'espace d'adressage mémoire global partitionné. Ce modèle propose une vision distribuée de la mémoire physiquement partagée par les threads. Ce modèle suggère donc la création d'un espace d'adressage virtuel partitionné global, auquel chaque thread a physiquement accès, mais dont les traitements sont partitionnés pour chaque thread. Ce modèle permet donc d'éviter, en grande partie, les problèmes de concurrence d'accès aux données, que l'on peut trouver dans tous les modèles basés sur le paradigme à verrous. Ce modèle de programmation est donc basé sur le paradigme de parallélisme de données, et s'implémente généralement par la parallélisation d'un traitement sur un tableau ou un conteneur. Notons enfin que, dans le modèle PGAS, le partitionnement proposé pour le traitement de données peut changer aucours de l'exécution du programme parallèle.

Mémoire distribuée. Le modèle de programmation parallèle *Message Passing Interface* (MPI) [61] est basé sur la paradigme de passage de messages et définit un protocole de communication entre des processus indépendants et éventuellement distants. Ce modèle a initialement été défini pour les architectures à mémoire distribuée, toutefois il obtient également de très bonnes performances sur des architectures à mémoire partagée et à mémoire hybride distribuée/partagée. Ce modèle est composé de communications *point-à-point*, permettant de décrire l'envoi d'un message à un processeur précis, et de communications *collectives*, permettant d'envoyer des informations à l'ensemble ou à une sous-partie des autres processus. Notons que les communications point-à-point peuvent être bloquantes ou non bloquantes pour permettre certaines optimisations dans les programmes parallèles implémentés. Une communication non bloquante rendra la main avant que la communication ne soit terminée, à l'inverse d'une communication bloquante. Il est alors à la charge de l'utilisateur de s'assurer, aux endroits adéquats de son programme, que la communication est terminée. MPI contient également des interfaces permettant de créer des topologies entre les processus, de créer des types d'envois particuliers etc. Il existe plusieurs implémentations génériques de cette norme, comme Open MPI [56] ou MPICH2 [63], et il est de plus possible pour les constructeurs d'écrire leur propre implémentation afin de l'optimiser à leur matériel. C'est notamment le cas de Intel MPI [22]. Ce modèle (et ses implémentations) a connu un grand succès depuis sa création dans les années 90, et s'impose aujourd'hui comme l'un des outils de référence de la parallélisation, tout particulièrement dans le domaine du calcul scientifique et de la haute performance.

Il est également important, pour la compréhension de cette thèse, de s'attarder sur l'un des modèles de programmation les plus connus et les plus anciens, le modèle *Bulk*

Synchronous Parallel [92, 118], proposé par Valiant en 1990. Le modèle architectural de BSP correspond naturellement à une machine à mémoire distribuée. En effet, dans ce modèle la machine modélisée est une machine à mémoire distribuée composée d'un ensemble de processeurs à mémoire indépendante. Toutefois, comme MPI, ce modèle peut tout à fait s'appliquer à une architecture à mémoire partagée. Les caractéristiques d'une machine BSP sont définies par quatre paramètres. Le premier, p, représente le nombre de processeurs sur la machine. Le deuxième, r, représente la puissance d'un processeur (mesurée en nombre d'opérations flottantes par seconde). L représente, quant à lui, le temps nécessaire pour effectuer une synchronisation globale entre tous les processeurs. Et pour finir, g représente le temps nécessaire pour l'envoi d'une donnée, du type souhaité, sur le réseau. L'élément de base d'un algorithme ou d'un programme BSP est appelé une *super-étape* (ou *superstep*). Un programme BSP est constitué d'une succession de super-étapes qui peuvent être composées (1) de plusieurs phases de calculs indépendantes, (2) de phases de communications, et (3) de phases de synchronisation entre les processeurs. On distingue plus généralement des super-étapes de calculs, dans lesquelles chaque processeur effectue une séquence d'opérations sur des données locales, et des super-étapes de communications, où chaque processeur envoie et reçoit des messages. Quelle que soit la représentation d'une super-étape, elle est toujours terminée par une synchronisation des processeurs. Dans cette phase de synchronisation chaque processeur vérifie que l'ensemble des tâches à accomplir sont terminées localement, et préviens les autres processeurs. Tous les processeurs attendent les messages de terminaison des autres processeurs avant que la super-étape ne se termine et qu'une autre puisse être commencée. Ce type de synchronisation est appelé *bulk synchronisation*. L'une des forces du modèle BSP est de proposer une fonction de coût calculée à partir des paramètres de la machine et de l'algorithme BSP formulé en super-étapes. Étant donné une super-étape de calcul s, on note $\omega^{(s)}$ le temps d'exécution de la super-étape, qui est égal au temps maximum d'exécution, parmi tous les processeurs. Nous avons alors $\omega^{(s)} = \max_{0 \leq i < p} \omega_i^{(s)}$. Étant donné un programme BSP contenant N_{comp} super-étapes de calcul, nous notons alors W le temps de calcul du programme, tel que $W = \sum_{s=0}^{N_{comp}} \omega^{(s)}$. Étant donné maintenant une super-étape de communication s, on note $h_i^{(s)}$ le maximum entre le nombre de données envoyées et reçues pendant la super-étape s et sur le processeur i. Nous définissons alors $h_i^{(s)} = max(h_{r_i}^{(s)}, h_{s_i}^{(s)})$, où $h_{r_i}^{(s)}$ et $h_{s_i}^{(s)}$ représentent respectivement le nombre de données reçues et envoyées par le processeur i, à la super-étape de communication s. On note alors $h^{(s)}$ le nombre de données en échange dans la super-étape s, qui est égal à $h^{(s)} = \max_{0 \leq i < p} h_i^{(s)}$. Étant donné un programme BSP contenant N_{comm} super-étapes de communication, nous notons finalement H la quantité de données échangées dans le programme BSP, tel que $H = \sum_{s=0}^{N_{comm}} h^{(s)}$. Nous obtenons alors la fonction de coût suivante

$$T = W + H \times g + (N_{comp} + N_{comm}) \times L.$$

Il est donc possible en écrivant un programme BSP de pouvoir obtenir une prévision du temps d'exécution du programme.

Plusieurs implémentations du modèle BSP ont été effectuées. On peut noter la bibliothèque BSPLib [69] écrite en C/Fortran, ou sa variante BSPonMPI [17]. Enfin notons MulticoreBSP [130] qui a été spécifiquement développée pour les architectures à mémoire partagée dans les CPU.

Enfin, nous pouvons évoquer quelques autres modèles de programmation issus du paradigme de passage de messages et induits par les architectures à mémoire distribuée, mais que nous n'utiliserons pas dans cette thèse. Par exemple, les modèles proposant un concept d'appel de procédures à distance RPC [16] (pour *Remote Procedure Call*) ou aussi d'invocation de méthodes à distance RMI (pour *Remote Method Invocation*). Ces modèles peuvent proposer des appels de procédures proche d'un appel local et séquentiel, et permettent des appels non bloquants afin de mettre en place le parallélisme. Le concept de variable *future* est introduit pour évoquer le résultat de l'appel à une procédure distante. L'utilisation d'une telle variable bloque alors l'exécution jusqu'à terminaison de la procédure distante. De même nous pouvons évoquer les modèles permettant l'appel parallèle de procédures, que l'on retrouve notamment dans les modèles de composants [14, 15].

Mémoire hétérogène. Nous avons vu qu'il était possible de coupler des architectures à mémoire partagée avec des architectures à mémoire distribuée. Ainsi, afin de pouvoir programmer ce type d'architectures, dites hybrides ou hétérogènes, il est également possible de coupler les modèles de programmation énoncés plus haut. Il est donc possible d'écrire des programmes en mélangeant l'utilisation de MPI ou BSP avec OpenMP, ou avec les pthreads, par exemple. Nous n'avons toutefois pas encore évoqué des modèles de programmation permettant de programmer sur architectures GPU, ou hybrides GPU et CPU. Le modèle *Computer Unified Device Architecture* (CUDA) [107] est proposé par le constructeur de GPU nVidia afin de programmer des calculs généraux sur les accélérateurs graphiques. CUDA peut notamment s'utiliser en C, C++ ou Fortran. Si CUDA est une référence de la programmation sur GPU, et une référence du calcul haute performance, il n'en reste pas moins un modèle de programmation parallèle complexe à mettre en œuvre. Il est, en effet, à la charge du programmeur CUDA d'effectuer explicitement les échanges de données de la mémoire principale vers la mémoire locale des GPU, mais aussi de définir l'organisation des processus légers créés, et le découpage des noyaux de calcul. Pour produire des programmes hybrides ou hétérogènes, il est également possible de mélanger MPI et CUDA, par exemple.

Le modèle de programmation proposé par OpenACC [128] permet la mise en place de programmes parallèles sur architectures hétérogènes CPU/GPU. Son fonctionnement est similaire à OpenMP, avec la mise en place de directives de parallélisation d'un plus haut niveau d'abstraction. Contrairement au modèle proposé par CUDA, OpenACC cache intégralement les détails liés à l'architecture cible à l'utilisateur. Ainsi le découpage en threads ainsi que les transfert de données d'une mémoire à l'autre sont cachés à l'utilisateur. Ce modèle permet donc une programmation GPGPU très facilitée et très intéressante.

Enfin, OpenCL [112] est un modèle de programmation qui propose d'écrire un code parallèle portable sur plusieurs architectures matérielles : les CPU et les GPU que nous

avons évoqués précédemment, mais également les DSP (*digital signal processor*), qui sont des micro-processeurs spécialisés pour les traitements numériques du signal, et les FPGA (*field-programmable gate array*), qui sont des circuits programmables à l'aide d'un langage de description matériel. OpenCL est donc un modèle qui aborde un autre problème de la programmation parallèle : la portabilité des codes. Étant donné la difficulté de programmation des codes parallèles et l'impact important que cela représente, notamment pour les scientifiques, cette problématique est très intéressante et peu traitée au sein des modèles. Notons toutefois que les modèles présentés précédemment, et comme déjà précisé, peuvent pour la plupart être implémentés pour différents modèles d'exécution, même si ils ne traitent pas directement le problème de portabilité.

2.2 Problèmes numériques et Simulations scientifiques

L'analyse numérique est une discipline des mathématiques qui s'intéresse aux fondements théoriques et à la recherche de méthodes pour résoudre des problèmes de l'analyse mathématique, de façon purement numérique [73]. Cette discipline est très largement utilisée afin de produire des codes de calcul, comme en météorologie par exemple, mais également afin d'effectuer des simulations de phénomènes réels, comme en aéronautique, hydrologie ou médecine, par exemple. Cette discipline est, de plus, en lien étroit avec les sciences informatiques. En effet, si sa partie théorique relève des mathématiques, sa mise en pratique, elle, conduit à la production d'algorithmes et de programmes. L'analyse numérique produit des programmes souvent compliqués à mettre en œuvre et nécessitant de bonnes connaissances de programmation. Grâce à une bonne connaissance de l'informatique et de la programmation, il est possible d'obtenir des programmes rapides et peu coûteux en mémoire, voire des programmes parallèles. C'est pour cette raison que, bien souvent, l'analyse numérique mène à des collaborations entre mathématiciens et informaticiens. L'analyse numérique consiste à développer des méthodes numériques permettant d'obtenir des solutions exactes ou approchées à des problèmes mathématiques. Les solutions approchées résultent, dans la plupart des cas, de méthodes de discrétisation, comme dans le cas des équations différentielles. Dans cette thèse nous nous intéressons plus particulièrement à la résolution d'équations aux dérivées partielles (EDP). Cette thèse s'intéresse, en effet, à proposer des solutions de parallélisme implicite permettant de résoudre des EDP par les méthodes de discrétisation, aussi appelées *méthodes basées sur des maillages*.

2.2.1 Équations aux dérivées partielles

Une équation différentielle est une relation mathématique entre une ou plusieurs fonctions et leurs dérivées. On distingue les équations différentielles ordinaires (EDO), dans lesquelles les fonctions et leurs dérivées ne dépendent que d'une variable, et les équations aux dérivées partielles (EDP), dans lesquelles les fonctions et leurs dérivées dépendent de plusieurs variables. Les EDP ont la particularité, comme leur nom l'indique,

de représenter une relation entre une ou plusieurs fonctions et leurs dérivées partielles. En effet, soit une fonction $u(t, x)$ qui dépend de deux variables t et x, il est possible de dériver cette fonction suivant la variable t ou suivant x uniquement. On note alors la dérivée partielle de u suivant t, $u_t = \frac{\partial u}{\partial t}$, et la dérivée partielle de u suivant x, $u_x = \frac{\partial u}{\partial x}$. Les EDP admettent souvent plusieurs solutions, les conditions d'une EDP étant moins strictes que dans une EDO, qui utilise une seule variable. Il est également souvent difficile de trouver numériquement une solution d'une EDP. Afin de restreindre le spectre de solutions d'une EDP, une EDP est associée à une ou plusieurs conditions aux limites du domaine étudié. L'EDP peut être considérée comme l'ensemble d'une équation et de ses conditions limites. En guise d'exemple, l'équation (2.1) est une EDO qui représente la troisième loi de Newton, et qui signifie que l'accélération d'un corps est proportionnelle aux forces qui lui sont appliquées. L'équation (2.2) est une EDP qui représente l'équation de la chaleur en deux dimensions. Dans la plupart des méthodes de résolution numérique des EDP, et comme nous allons le voir dans cette partie, une discrétisation des variables est utilisée, ce qui revient à exprimer la solution de l'EDP par un ensemble fini de points et d'intervalles.

$$m\frac{d^2x(t)}{dt^2} = -F(x(t)) \tag{2.1}$$

$$\frac{\partial u(x,y,t)}{\partial t} = \frac{\partial^2 u(x,y,t)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \tag{2.2}$$

2.2.2 Passage du continu au discret

2.2.2.1 Discrétisation

Un ordinateur n'étant pas capable de résoudre une équation sur des variables continues, les simulations d'EDP sont très souvent transformées en problèmes discrétisés. Dans le cas d'une EDP, deux types de dimensions sont souvent représentés :

— le premier est le temps. Cette dimension sert à modéliser l'évolution du phénomène réel dans le temps. Sa discrétisation consiste alors à découper le temps de la simulation en un pas constant, que l'on note Δt, et qui est calculé de façon à rester cohérent avec l'éventuel mouvement calculé dans la simulation ;

— le deuxième type de dimension représenté est le domaine physique dans l'espace, ou plus simplement, le domaine, noté Ω. Suivant que la simulation représente un phénomène dans une, deux ou trois directions (ou dimensions en espace), les EDP feront intervenir une, deux, ou trois variables supplémentaires au temps, que l'on note x, y, et z. La discrétisation du domaine d'étude Ω, et donc des variables associées (x, y, z), est alors appelée un *maillage*.

En théorie, donc, on dit qu'une EDP est de dimension 2 lorsque elle dépend de t et de x, ou de dimension 4 lorsqu'elle dépend de t et (x, y, z). Toutefois, on ne dénote généralement, et dans le reste de cette thèse, que les dimensions en espace pour décrire une EDP. La dimension suivant le temps est, en effet, généralement sous entendue. L'équation de Laplace présentée dans l'équation (2.2), par exemple, est une EDP à deux dimensions, ou 2D.

2.2.2.2 Maillages

Un maillage est issu de la discrétisation d'une ou plusieurs variables spatiales dans une EDP. Il s'agit d'un ensemble de points reliés entre eux et qui forment le domaine d'étude de façon simplifiée et discrétisée. Un maillage est plus précisément défini par (1) un repère, (2) les points qui le constituent (tous associés à des coordonnées), et (3) les cellules formées en reliant ces points. Un maillage est caractérisé par sa dimension (typiquement 1D, 2D ou 3D), son aire ou son volume (suivant qu'il est 2D ou 3D), sa finesse (précision du maillage), et la géométrie de ses cellules (triangles, carrés, tétraèdres, cubes etc.). Plus un maillage est fin plus l'aire moyenne (ou le volume moyen en 3D) représentée par ses cellules est faible. Plus le maillage est fin, plus on se rapproche du domaine continu et plus la simulation pourra être précise. Parmi tous les maillages possibles, on peut noter plus particulièrement deux grandes familles, les maillages structurés et non-structurés. Typiquement, on dit qu'un maillage est structuré si il peut être généré en reproduisant plusieurs fois la même maille. À l'inverse un maillage est non-structuré si chacune de ses mailles est différente.

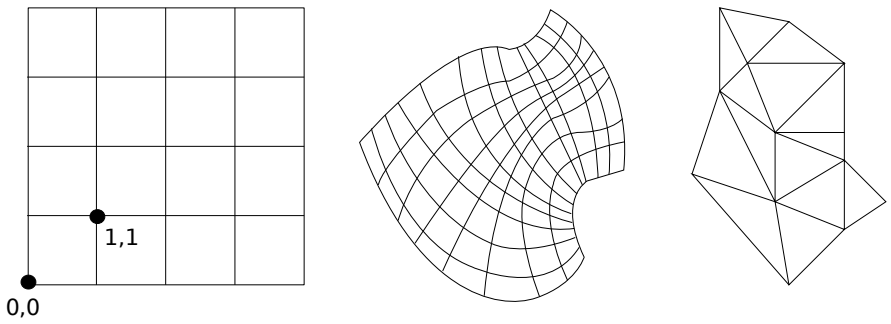


FIGURE 2.1 – *De la gauche vers la droite : maillage cartésien, curvilinéaire et non-structuré.*

Maillages structurés cartésiens. Ce type de maillage est régulier et rectangulaire et est illustré dans la figure 2.1. Historiquement, la discrétisation en points du domaine a tout d'abord été associée à des maillages structurés de type cartésiens dans lesquels il est facile de déterminer le voisinage d'un point. Toutefois ce n'est pas le seul type de grille régulière qu'il est possible d'utiliser pour une discrétisation.

Maillages structurés curvilinéaires. Il est également possible d'utiliser d'autres systèmes de coordonnées, comme des coordonnées cylindriques ou sphériques qui peuvent toujours être assimilés à des grilles structurées, mais déformées. Ce type de maillages, dit curvilinéaire, et représenté dans la figure 2.1, peut être considéré comme rectangulaire et n'est pas différent, du point de vue de la programmation, des maillages cartésiens. Le livre *Handbook of Grid Generation* [117] illustre bien cette caractéristique des maillages curvilinéaires en faisant un parallèle entre un maillage et une éponge. Imaginons une

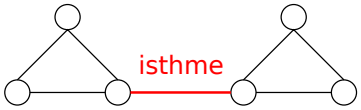
éponge rectangulaire sur laquelle a été dessinée une grille cartésienne régulière. Si l'on roule l'éponge de façon à ce que son bord droit rejoigne son bord gauche, on obtient une grille curvilinéaire cylindrique. Cette nouvelle grille peut donc être assimilée à une grille rectangulaire où deux colonnes "fantômes" ont été ajoutées à droite et à gauche afin de représenter la liaison des deux bords. Les distances séparant les points d'une grille rectangulaire et d'une grille curvilinéaire sont différentes et modifieront les schémas numériques, mais pas l'utilisation du maillage en lui-même. Il paraît bien sûr évident que l'éponge peut être déformée de façon à former n'importe quel objet, et que l'éponge peut également être étirée ou réduite. Toutefois, dans les cas complexes, l'éponge peut être tellement modifiée qu'il devient difficile de l'utiliser dans le schéma numérique. Dans ce cas, plusieurs éponges sont utilisées, chacune avec une transformation différente pour représenter l'espace. Chaque éponge est donc une grille curvilinéaire associée à une grille rectangulaire. Chaque éponge est alors appelée un *bloc* et ce type de grilles est appelé une *grille de blocs*. Les blocs doivent alors être reliés entre eux par l'ajout de mailles fantômes et la gestion des conditions limites, ce qui peut rendre la résolution compliquée.

Maillages non-structurés. Un maillage non-structuré est constitué d'un ensemble de points qui forment un ensemble de cellules. Ces cellules représentent les même géométries mais sont de dimensions différentes. Tout comme les maillages en blocs, les maillages irréguliers permettent de représenter des domaines de formes diverses et variées. Toutefois aucune transformation de la grille vers une grille rectangulaire cartésienne n'est effectuée. Dans ce cas, c'est le maillage en lui-même qui est complexe. Il est constitué d'un ensemble de points, ayant chacun des coordonnées dans l'espace (2D, 3D, etc.), et reliés entre eux de façon à former des faces et des cellules. Généralement, les maillages non-structurés sont constitués de triangles (en 2D) ou de tétraèdres (en 3D), et l'ensemble de ces cellules peut former la surface ou le volume d'un objet complexe. Un exemple de maillage non-structuré est donné dans la figure 2.1. Plus les mailles sont petites, plus la représentation sera fidèle et précise. L'une des différences majeures entre un maillage structuré et un maillage non-structuré est sa représentation en tant que structure de données informatique. Un maillage structuré à deux dimensions peut naturellement être associé à une matrice, ce qui simplifie sa programmation, les accès aux voisinages et son partitionnement. A l'inverse, dans un maillage non-structuré, il est nécessaire par un moyen ou un autre de représenter une carte des connectivités entre les éléments. Ces maillages sont très largement utilisés dans les simulations les plus complexes.

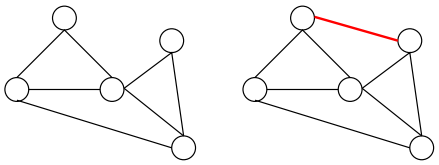
Maillages hybrides. Enfin, notons que des maillages hybrides, peu développés à l'heure actuelle, sont susceptibles de connaître un essor. Ils consistent à définir plusieurs maillages réguliers reliés entre eux par un maillage irrégulier. Tout comme les maillages constitués de plusieurs grilles curvilinéaires (grille de blocs), les maillages hybrides sont très complexes, et difficiles à mettre en œuvre.

Représentation en graphes. En théorie des graphes, un maillage est un graphe $G = (V, E)$ (avec V l'ensemble des sommets et E l'ensemble des arêtes du graphe) projeté sur

$\mathbb{R}$, $\mathbb{R}^2$, $\mathbb{R}^3$ ou davantage de dimensions. En d'autres termes, un maillage est un graphe dont chaque nœud est associé à des coordonnées dans $\mathbb{R}^n$. Deux caractéristiques peuvent être isolées dans un graphe représentant un maillage. Tout d'abord, le graphe $G = (V, E)$ associé à un maillage est un graphe connexe, c'est-à-dire que pour tout nœud u et $v \in V$ il existe une suite d'arêtes les reliant. De plus, un graphe représentant un maillage est un graphe sans isthme. Dans un graphe, un isthme est un pont dont l'élimination induit la formation de deux composantes connexes (c'est-à-dire que le graphe n'est plus connexe). Cette caractéristique garantie que les nœuds du graphe forment des cellules, et est illustrée dans la figure 2.2(a). Un graphe représentant un maillage pourrait également être considéré comme un graphe uniquement composé de cycles élémentaires, mais cette définition irait à l'encontre du type de graphe illustré à gauche de la figure 2.2(b), qui est a priori un cas envisageable pour un maillage. Le graphe équivalent composé uniquement de cycles élémentaires serait alors le graphe représenté à droite de la figure 2.2(b). Notons, enfin que la planarité d'un maillage sur $\mathbb{R}$ ou $\mathbb{R}^2$ est évidente mais qu'elle n'est pas garantie sur $\mathbb{R}^3$.



(a) Exemple de graphe contenant un isthme et ne représentant donc pas un maillage.



(b) De gauche à droite : graphe représentant a priori un maillage et n'étant pas constitué que de cycles élémentaires, et son équivalent constitué uniquement de cycles élémentaires

FIGURE 2.2 – *Maillages en théorie des graphes.*

2.2.3 Méthodes numériques basées sur les maillages

2.2.3.1 Précision, stabilité, convergence et ordre d'une méthode

Afin de connaître la précision d'un résultat numérique, il est idéalement nécessaire de contrôler les erreurs qui se produisent lors du calcul. Elles peuvent être dues à des limitations matérielles ou à des approximations dans les solutions appliquées.

Précision. Dans les calculs peuvent se produire des *erreurs d'arrondi*. Ces erreurs sont imposées par le matériel informatique où la représentation d'un nombre en mémoire est finie. Par conséquent, un nombre réel n'est connu qu'avec une précision donnée n telle que

$$x = a + 0, a_1 a_2 ... a_n$$

où a et a_i, pour tout i dans $[\![1, n]\!]$, sont des entiers naturels. Lorsque les nombres flottants sont manipulés lors d'un calcul, ils peuvent être soumis à des arrondis. Ces erreurs

d'arrondi, si elles ne sont pas contrôlées, peuvent conduire à des erreurs importantes dans les calculs numériques. En effet, les erreurs d'arrondi, dans les processus itératifs ou récurrents s'ajoutent, ce qui peut conduire à une forte amplification d'une erreur au départ négligeable. De plus, l'erreur peut être remarquée à un endroit inattendu si elle est répercutée dans différents endroits du code.

Stabilité, convergence et ordre. Les méthodes numériques basées sur des maillages sont des méthodes basées sur des approximations. En effet, un maillage en lui-même est déjà une approximation du domaine, mais les méthodes numériques font également des approximations, que nous détaillerons par la suite pour trois exemples. Les méthodes numériques ne font donc qu'approcher la solution exacte du problème et sont par définition basées sur des erreurs. La *stabilité* garantit que cette erreur ne s'amplifie pas, afin que la simulation converge vers une solution. Soit I un intervalle de $\mathbb{R}$ et soit u une solution à une EDP telle que $u : I \to \mathbb{R}$. L'intervalle $I = [a, b]$ est discrétisé de telle sorte que $a = x_0 < x_1 < ... < x_n = b$. On note $h_i = x_i - x_{i-1}$ et $h = \sup_{1 \leq i \leq n} (h_i)$. Soit v une solution approchée de u. On appelle *erreur de consistance*, pour l'élément x_i de la discrétisation, la quantité $e_i = u(x_i) - v(x_i)$ et l'erreur globale l'expression

$$e = \sup_{0 \leq i \leq n} |u(x_i) - v(x_i)|$$

On dit que la méthode converge si $\lim_{h \to 0} e = 0$. Par conséquent on dira qu'une méthode converge si la diminution du pas h réduit l'erreur e et rend ainsi la solution plus précise. Bien évidemment, plus h est petit, plus il y a de calculs à faire, car alors la discrétisation est plus fine et donc le nombre de points ou de cellules à calculer plus grand. Il s'agit donc de trouver le juste milieu entre un calcul trop long et un calcul qui diverge trop de la solution exacte. Le schéma de discrétisation v de u est d'ordre p si $e = O(h^p)$. En supposant que $h < 1$, alors plus l'ordre est grand, plus l'erreur est petite et la méthode précise.

2.2.3.2 Méthodes implicites et explicites

Lorsque les variables d'une EDP sont discrétisées et que l'une de ces variables représente le temps, l'EDP se transforme en un schéma de discrétisation, ou schéma numérique, dont le rôle est de calculer pour chaque nouveau pas de temps t la valeur de la solution. Si ce schéma dépend des états connus de la solution aux itérations de temps précédentes $t-1$, $t-2$ etc., alors le schéma est dit *explicite*. En revanche, si le schéma dépend d'états encore inconnus de la solution à t, le schéma est dit *implicite*. Des dépendances entre les éléments d'une même itération sont alors créées, posant des problèmes d'ordonnancement. Notons μ le maillage issu de la discrétisation des variables en espace, E_μ un ensemble d'éléments du maillage μ, qui peut représenter par exemple les points, les arêtes, ou le centre des mailles etc., T l'ensemble des itérations de la simulation issu de la discrétisation en temps. Soit V l'ensemble des quantités de la simulation (par exemple la pression, la vitesse etc.), nous notons alors σ la fonction $\sigma : E_\mu \times T \to V$, qui permet d'obtenir

l'état de la simulation, c'est-à-dire l'ensemble des valeurs des quantités simulées, pour un élément de μ et à un instant t. Un schéma numérique explicite est généralement de la forme

$$\{\sigma(x,t-1),\sigma(y,t-1);y\in N(x)\}\longmapsto\sigma(x,t) \tag{2.3}$$

où $x\in E_\mu$, $t\in T$, et où $N(x)\subset E_\mu$ représente le voisinage nécessaire au calcul autour du point (ou de la cellule) x, aussi appelé *stencil*. Cette relation signifie que le calcul de $\sigma(x,t)$ s'effectue en fonction des quantités calculées à l'itération de temps précédente $t-1$ pour le même élément x, et pour un voisinage de cet élément, noté $N(x)$. Un schéma numérique implicite est, quant à lui, généralement de la forme

$$\{\sigma(x,t-1),\sigma(y,t-1),\sigma(y,t);y\in N(x)\}\longmapsto\sigma(x,t). \tag{2.4}$$

Dans un schéma numérique implicite, une dépendance $\sigma(y,t)$ est créée et rend la programmation de la solution plus complexe. Toutefois un schéma implicite est souvent plus stable qu'un schéma explicite. Pour des raisons de simplicité dans le codage des simulations, l'idéal est d'obtenir un ou plusieurs schémas explicites stables.

2.2.4 Exemples de méthodes numériques basées sur des maillages

2.2.4.1 Méthode des différences finies

La méthode des différences finies est la méthode la plus ancienne qui permet de passer d'un ensemble d'équations aux dérivées partielles continues à un ensemble de schémas numériques, implémentables sur un ordinateur. Historiquement, cette méthode est associée à une discrétisation régulière de type cartésienne du domaine, toutefois il est possible de l'associer à des maillages curvilinéaires et à des maillages en blocs pour les cas les plus complexes. Nous allons décrire ici son principe pour le cas le plus simple associé à la création de maillage cartésien. Cette partie s'inspire du document de De Sterck et Al [110] et du livre de Pinchover et Al [102].

La méthode des différences finies est basée sur le *théorème de Taylor* aussi appelé la *formule de Taylor*.

Théorème 1 *Soit I un intervalle de $\mathbb{R}$ et $a\in I$. Soit E un espace vectoriel normé, et soit $f:I\to E$ une fonction n fois dérivable en a. Alors pour tout $x\in I$*

$$f(x)=f(a)+\frac{f^{(1)}(a)}{1!}(x-a)+\frac{f^{(2)}(a)}{2!}(x-a)^2+\ldots+\frac{f^{(n)}(a)}{n!}(x-a)^n+o((x-a)^n)$$

où $f^{(n)}$ est la dérivée n-ième de f, et où $o((x-a)^n)$ désigne une fonction négligeable devant $(x-a)^n$.

En d'autres termes, une fonction plusieurs fois dérivable au voisinage d'un point peut être approximée par une fonction polynôme dont les coefficients dépendent uniquement des dérivées de la fonction en ce point.

Pour illustrer l'utilisation de ce théorème dans la méthode des différences finies, prenons un cas simple en une dimension. On considère un domaine $\Omega = \{x : x \in [0,1]\}$, et l'on cherche à déterminer une solution u telle que

$$u^{(2)}(x) = f(x), \tag{2.5}$$

pour tout $x \in \Omega$. On discrétise alors le domaine Ω en $m+2$ éléments distincts $x_0, x_1, \ldots, x_{m+1}$, où $x_0 = 0$ et $x_{m+1} = 1$ sont les points extrêmes du domaine, et $x_1, \ldots, x_m$ les points internes au domaine (figure 2.3). On considère chaque intervalle de cette discrétisation comme régulier tel que $x_i - x_{i-1} = \Delta x = h$.

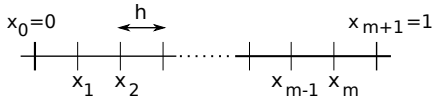


FIGURE 2.3 – *Discrétisation régulière de* $\Omega = \{x : x \in [0,1]\}$.

Dans la suite, on utilisera la notation $u_i = u(i)$. En appliquant le théorème de Taylor à u aux points $x = i+1$ puis $x = i-1$ pour $a = i$, nous avons

$$u_{i+1} = u_i + u_i^{(1)}h + \frac{1}{2}u_i^{(2)}h^2 + \frac{1}{6}u_i^{(3)}h^3 + \frac{1}{24}u_i^{(4)}h^4 + o(h^4)$$

$$u_{i-1} = u_i - u_i^{(1)}h + \frac{1}{2}u_i^{(2)}h^2 - \frac{1}{6}u_i^{(3)}h^3 + \frac{1}{24}u_i^{(4)}h^4 + o(h^4).$$

Ainsi, en sommant u_{i+1} et u_{i-1}, nous obtenons

$$u_{i+1} + u_{i-1} = 2u_i + u_i^{(2)}h^2 + \frac{1}{12}u_i^{(4)}h^4 + o(h^4)$$

d'où

$$u_i^{(2)} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2} - \frac{1}{12}u_i^{(4)}h^2 + o(h^2)$$

La méthode des différences finies consiste alors à chercher une approximation de la solution u de l'équation (2.5), notée v telle que :

$$\frac{v_{i+1} - 2v_i + v_{i-1}}{h^2} = f_i \quad \forall i \in [\![0, m+1]\!] \tag{2.6}$$

car $-\frac{1}{12}u_i^{(4)}h^2 + o(h^2)$ est négligeable si h est petit.

On illustre donc ici qu'en discrétisant l'espace et en utilisant le théorème de Taylor, on obtient une solution approchée de u qui s'exprime par des expressions numériques. L'équation (2.6) est appelée un schéma numérique.

Considérons maintenant un cas simple en deux dimensions. Soit $\Omega = \{(x,y) : (x,y) \in [0,1]^2\}$ le domaine étudié. On cherche à déterminer une solution u de l'équation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y). \tag{2.7}$$

En discrétisant le domaine en $m+2$ éléments dans les deux dimensions et tel que $\Delta x = \Delta y = h$ (figure 2.4) et en appliquant le théorème de Taylor à l'ordre 4, comme précédemment, on obtient pour i et j dans $[\![0, m+1]\!]$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + o(h^2)$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} + o(h^2),$$

où $u_{i,j}$ désigne $u(i,j)$. Soit v une approximation d'une solution exacte u de l'équation (2.7). On obtient alors le schéma numérique suivant

$$\frac{v_{i+1,j} + v_{i-1,j} - 4v_{i,j} + v_{i,j+1} + v_{i,j-1}}{h^2} = f_{i,j} \quad \forall\ (i,j) \in [\![0, m+1]\!] \times [\![0, m+1]\!] \quad (2.8)$$

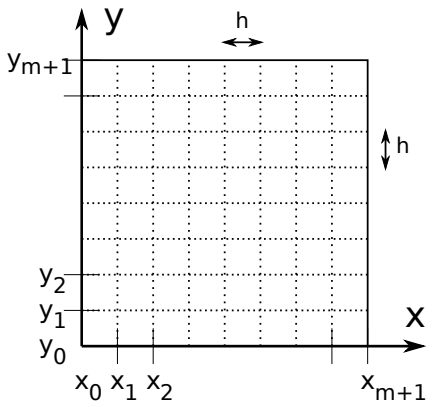


FIGURE 2.4 – *Discrétisation régulière de* $\Omega = \{(x,y) : (x,y) \in [0,1]^2\}$.

Un schéma numérique exprime le voisinage nécessaire, dans le domaine discrétisé, pour permettre la résolution du système d'EDP. Ce voisinage est aussi appelé *stencil* et sera un terme utilisé dans le reste de ce manuscrit. Lorsqu'une variable représentant le temps est ajoutée aux systèmes d'équations à résoudre, une dimension est de nouveau ajoutée, et une discrétisation de cette variable doit être effectuée à son tour. Le schéma numérique est alors exprimé en fonction de la discrétisation en espace et de la discrétisation en temps. Comme expliqué précédemment, si le schéma numérique exprime la solution à l'instant $t+1$ en fonction des instants précédents, la méthode est explicite (2.3). Si, en revanche, le schéma numérique exprime la solution à l'instant $t+1$ en fonction des instants précédents et de l'instant en cours de calcul, la méthode est implicite (2.4).

2.2.4.2 Méthode des volumes finis

Les méthodes des volumes finis sont des méthodes permettant de résoudre des équations qui peuvent être formulées grâce aux lois de conservation. Ces méthodes sont plus

complexes que les méthodes des différences finies. En effet, si les méthodes des différences finies se basent sur les points de la discrétisation, ou du maillage, pour obtenir des schémas numériques discrets, les méthodes des volumes finis se basent, quant à elles, sur les cellules formées par les points du maillage. Ces méthodes étudient, en effet, la valeur moyenne de la solution approchée sur un ensemble de cellules. Nous définissons l'ensemble de ces méthodes, plus généralement, comme "la" méthode des volumes finis, et une explication générale de cette méthode est présentée dans cette partie. Cette partie s'inspire, de nouveau, du document de De Sterck et Al [110] et du livre de Pinchover et Al [102].

Définition 1 *La forme différentielle d'une équation de conservation 1D pour une quantité $u(x,t)$ est donnée par*

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \tag{2.9}$$

où $f(u)$ est appelée la fonction de flux.

La première forme intégrale de la loi de conservation intègre l'équation (2.9) suivant un intervalle $[a,b]$ représentant l'unique dimension du problème selon x. En appliquant le fait que l'intégrale suivant x de la dérivée partielle suivant x de la fonction f est égale à f, la première forme intégrale de la loi de conservation s'écrit alors

$$\frac{d}{dt} Q(t) = f(u(a,t)) - f(u(b,t)), \tag{2.10}$$

où

$$Q(t) = \int_a^b u(x,t) \mathrm{d}x$$

La deuxième forme intégrale de la loi de conservation est obtenue en intégrant de nouveau la première forme intégrale (2.10) suivant un intervalle de temps $t \in [0,T]$. En appliquant de nouveau la même loi sur l'intégrale d'une dérivée partielle suivant t, nous obtenons

$$Q(T) - Q(T_0) + \int_0^T [f(u(a,t)) - f(u(b,t))] \mathrm{d}t = 0 \tag{2.11}$$

Une interprétation de l'équation (2.11) est illustrée dans la figure 2.5. Cette interprétation consiste à dire que le volume de la solution u sur l'intervalle $[a,b]$ entre les temps 0 et T est égal à la différence des flux entrants et sortants en a et b intégrés sur $[0,T]$.

Afin de mettre en œuvre la méthode des volumes finis, cette seconde forme intégrale est réécrite après discrétisation du domaine en cellules. Considérons une discrétisation sur l'espace à une dimension en cellules de même taille Δx, et une discrétisation du temps. On obtient une discrétisation illustrée dans la figure 2.6. La cellule dont le point central est i est alors représentée entre $x_{i-\frac{1}{2}}$ et $x_{i+\frac{1}{2}}$. Après discrétisation, on note $Q(x_i,t_n) = Q_i^n$. La deuxième forme intégrale de la loi de conservation à une dimension peut alors s'écrire

$$Q_i^{n+1} - Q_i^n + \int_{t_n}^{t_{n+1}} [f(u(x_{i+\frac{1}{2}},t)) - f(u(x_{i-\frac{1}{2}},t))] \mathrm{d}t = 0 \tag{2.12}$$

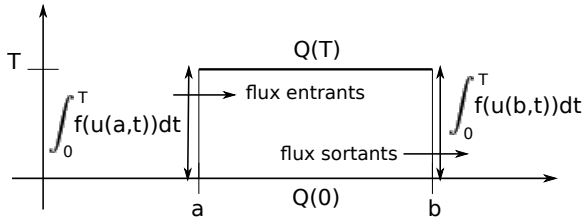


FIGURE 2.5 – *Interprétation de la seconde forme intégrale de la loi de conservation.*

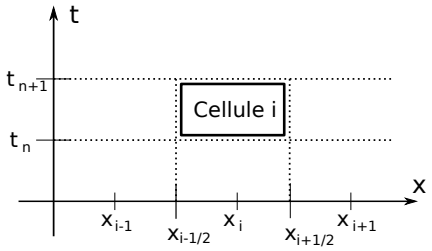


FIGURE 2.6 – *Discrétisation en cellules à volume fini suivant x et t.*

$$Q_i^n = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x,t_n)\mathrm{d}x.$$

Soit $\overline{u}_i^n$ la moyenne des valeurs de $u(x,t)$ sur la cellule i à t_n :

$$\overline{u}_i^n = \frac{Q_i^n}{\Delta x}$$

et soit $\overline{f}_{i+\frac{1}{2}}^{n+\frac{1}{2}}$ la moyenne des valeurs de $f(u)$ à l'interface $i+\frac{1}{2}$ entre t_n et t_{n+1} :

$$\overline{f}_{i+\frac{1}{2}}^{n+\frac{1}{2}} = \frac{\int_{t_n}^{t_{n+1}} f(u(x_{i+\frac{1}{2}},t))\mathrm{d}t}{\Delta t}.$$

La troisième forme intégrale de la loi de conservation est définie par :

$$\frac{\overline{u}_i^{n+1} - \overline{u}_i^n}{\Delta t} + \frac{\overline{f}_{i+\frac{1}{2}}^{n+\frac{1}{2}} - \overline{f}_{i-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta x} = 0. \tag{2.13}$$

En d'autres termes, la troisième forme intégrale fait une moyenne des quantités représentées dans la deuxième forme intégrale de la loi de conservation.

La méthode des volumes finis consiste à chercher une approximation de u, notée v, telle que

$$\overline{u}_i^n \approx v_i^n, \quad \overline{f}_{i+\frac{1}{2}}^{n+\frac{1}{2}} \approx f^*(v_i^n, v_{i+1}^n)$$

où $f^*(v_i^n, v_{i+1}^n)$ est appelée la fonction des flux numériques. On obtient alors, par la méthode des volumes finis, un schéma numérique explicite de la forme

$$\frac{v_i^{n+1} - v_i^n}{\Delta t} + \frac{f^*(v_i^n, v_{i+1}^n) - f^*(v_{i-1}^n, v_i^n)}{\Delta x} = 0 \tag{2.14}$$

C'est le choix de la fonction f^* qui différencie les différentes méthodes à volumes finis. Enfin, pour une loi de conservation à deux dimensions, l'idée de résolution est la même mais les fonctions de flux numériques sont alors au nombre de deux, et notées f^* et g^*.

2.2.4.3 Méthode des éléments finis

La méthode des éléments finis ne peut pas être décrite par une série de définitions et de formules comme pour les méthodes des différences et des volumes finis. En effet, cette méthode repose sur l'analyse des EDP, ce qui rend chaque cas particulier. Nous allons brièvement décrire le principe de cette méthode, sans entrer dans les détails car elle n'est pas utilisée dans les simulations mises en place dans cette thèse. Nous choisissons toutefois d'introduire cette méthode qui fait partie des méthodes traitées par d'autres solutions, auxquelles nous allons nous comparer, et qui fait partie des cas potentiels d'application de cette thèse.

Comme nous l'avons vu, la méthode des volumes finis consiste à discrétiser l'espace en un ensemble de points qui forment des cellules (graphe sans isthme), et à étudier, en chaque cellule, les flux entrants et sortants en se basant sur le principe de conservation des volumes. La méthode des éléments finis discrétise également l'espace en un ensemble de points formant des cellules. Mais cette méthode cherche, quant à elle, à approcher la solution exacte, en chaque cellule, à l'aide d'un ensemble de fonctions de base au préalable définies. La méthode des éléments finis se décompose en trois grandes étapes :

1. La discrétisation du domaine en N points formant des cellules. Pour effectuer cette discrétisation, la formulation dite "faible" des EDP est tout d'abord recherchée puis discrétisée. Cette formulation faible permet de chercher une solution continue par morceaux, et de toujours résoudre les EDP.
2. La définition de N fonctions de base Φ_j, pour $j \in [\![0, N[\![$, à utiliser, en chaque cellule, pour approcher la solution exacte. Ces fonctions sont exactes aux bords de la cellule, ce qui permet de retrouver la solution exacte aux points de la discrétisation.
3. La recherche des coefficients c_j d'une solution approchée v écrite sous la forme $v = \sum_{j=0}^{N-1} c_j \Phi_j$. Étant donné que la définition des N fonctions Φ_j a été effectuée au point (2), cette étape consiste à poser N équations linéaires permettant de déterminer l'ensemble des valeurs c_j.

Une visualisation simple de la solution exacte et de la solution approchée, obtenue pour un problème à une dimension, est illustrée par la figure 2.7.

Notons que suivant le type de fonctions Φ_j choisies, les méthodes diffèrent. Dans la méthode des éléments finis, on utilise communément des fonctions de type "tente" représentées sur le domaine $[a, b]$ dans la figure 2.8. Lorsque les fonctions utilisées représentent un spectre de vagues de différentes fréquences, on dit que la méthode est *spectrale* [82].

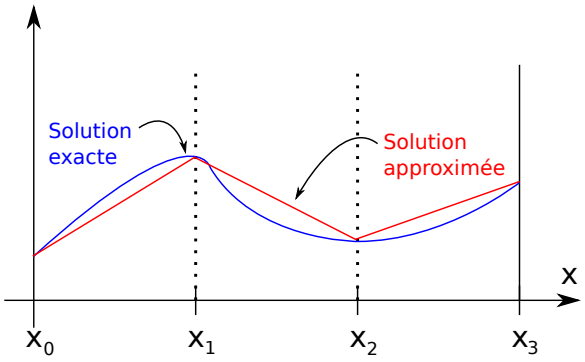


FIGURE 2.7 – *Illustration de la méthode des éléments finis pour un cas simple à une dimension.*

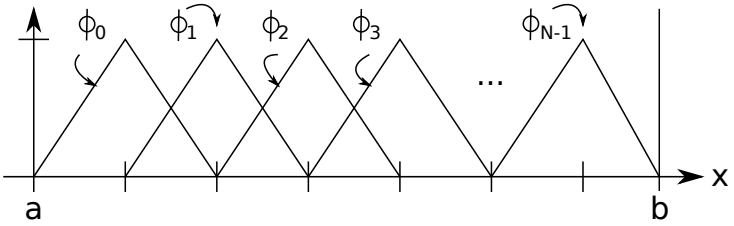


FIGURE 2.8 – *Fonctions de base Φ_j de type "tente".*

2.2.5 Programmation et parallélisation

Après avoir traité un ensemble de généralités pour comprendre les simulations scientifiques et leur problématiques, et après avoir brièvement expliqué les trois méthodes les plus connues de résolutions numériques, nous allons aborder leur programmation séquentielle, puis leur programmation parallèle. Dans cette thèse l'utilisation du terme simulation scientifique fera référence à des simulations dont le phénomène réel est modélisé par des EDP, et dont les méthodes de résolution sont basées sur des maillages. Nous nous limitons, de plus, et dans l'état actuel du travail, à l'obtention de schémas numériques explicites stables.

2.2.5.1 Programmation séquentielle et optimisations

La programmation d'une simulation modélisée par des EDP, dont la méthode de résolution est basée sur des maillages et sur des schémas numériques explicites, a toujours la même forme. Le pseudo-algorithme de la programmation séquentielle d'une simulation est présenté dans l'algorithme 1.

La programmation séquentielle d'une simulation possède donc deux grandes difficultés. La première est la programmation d'une structure de données permettant de représenter le maillage et sa connectivité. Afin que le programme soit suffisamment performant, cette structure de données doit permettre un accès rapide aux éléments du

Algorithme 1 : Algorithme séquentiel d'une simulation basée sur un maillage.

```
Création du maillage μ
Création des quantités à simuler appliquées à μ
Initialisation des quantités et des paramètres
Définition du pas de temps : Δt
Définition du temps maximal : tmax
tant que t < tmax faire
    pour chaque élément de la bordure physique : b faire
        Calcul des conditions limites de la simulation
    fin
    pour chaque x ∈ E_μ faire
        Obtention de σ(x, t − 1) et σ(y, t − 1), y ∈ N(x)
        Calcul de σ(x, t) en suivant le schéma explicite
    fin
    t = t + Δt
fin
```

maillage et également à leur voisinage, afin de calculer les *stencils*. La deuxième difficulté est ensuite d'effectuer l'ensemble des calculs numériques de la simulation de façon efficace. Cette efficacité dépend en grande partie de l'efficacité de la structure de données, mais elle dépend également de la programmation et du conditionnement des calculs numériques. Afin de rendre un calcul performant, il est tout d'abord important de réfléchir à l'utilisation de la mémoire cache de la machine.

Une bonne utilisation de cette mémoire cache réside dans le fait de favoriser la lecture d'éléments contigus en mémoire, et donc déjà chargés en cache, ainsi que de favoriser la réutilisation de données récemment calculées, elles aussi présentes en cache. Pour agir sur cette optimisation il est important de convenablement écrire les boucles sur les éléments du maillage, de réfléchir à la façon dont doivent être organisés les éléments de la structure de données et à l'organisation du calcul en lui-même.

La deuxième façon de rendre un calcul performant, qui peut paraître évidente, est de réduire au maximum son nombre d'opérations arithmétiques, en particulier les multiplications et divisions, plus coûteuses en temps d'exécution. La factorisation de boucles est une façon simple de réduire le nombre d'opérations dans un calcul, mais le calcul en lui-même peut être organisé différemment afin de réduire le nombre d'opérations. Prenons un exemple illustré par Jedrzejewski [73]. Imaginons que dans un programme nous soyons amenés à évaluer un polynôme $P(x) = ax^4 + bx^3 + cx^2 + dx + e$ en un point x. Si l'on calcule au préalable les variables suivantes

$$\begin{aligned} \alpha &= (b-a)/2a \\ \beta &= (d\alpha c/a^2) + \alpha^2(\alpha+1) \\ \gamma &= (c/a) - \alpha(\alpha+1) - \beta \\ \delta &= e - a\beta\gamma \end{aligned}$$

et que ces variables sont accessibles dans le reste du programme, alors l'évaluation du polynôme P en x ne coûte plus que trois multiplications :

$$P(x) = [(x + \alpha)x + \beta][(x + \alpha)x + (x + \gamma)] + \delta.$$

L'optimisation des codes séquentiels est une première étape pour obtenir des simulations performantes. Toutefois, même si les optimisations augmentent la performance des codes séquentiels, il est très souvent nécessaire, dans le domaine du calcul scientifique, de produire des programmes parallèles.

2.2.5.2 Parallélisation

Le domaine de la simulation scientifique est largement à l'origine des besoins en performances et en programmation parallèle. Ces besoins permanents de performances sont principalement dus à trois phénomènes :

1. Tout d'abord, plus les simulations sont complexes et précises, plus le nombre de calculs numériques à effectuer est grand. Le temps de calcul d'une simulation est d'autant plus long qu'il y a de calculs à effectuer.
2. La discrétisation du temps, ou autrement dit la "durée" simulée, peut également être à l'origine d'un temps de calcul important. En effet, si l'on simule une longue durée avec un pas de temps petit, le nombre d'itérations en temps à effectuer devient important. De plus, l'ensemble des calculs numériques doit être effectué pour chaque itération de temps supplémentaire.
3. Enfin, comme nous l'avons vu précédemment, plus la discrétisation du domaine en espace est fine, plus la solution est précise, et plus le nombre de calculs à effectuer est important.

Étant donné que plus un calcul est précis, plus les résultats sont intéressants pour la communauté, et que la précision d'un calcul peut être augmenté presque sans limite (avec une taille du domaine importante, une discrétisation du domaine très précise, et un temps simulés de plus en plus longs), le besoin en parallélisme est lui aussi sans limite pour le calcul scientifique. Il devient d'ailleurs limitant pour les scientifiques, en terme de résultats, de ne pas produire des codes parallèles. De nombreux travaux décrivent la parallélisation de problèmes et de simulations numériques.

Une première façon d'appréhender la parallélisation de simulations scientifiques, est de chercher, pour chaque sous-problème numérique posé (factorisation LU, méthode de Jacobi, interpolation etc.), une parallélisation idéale et optimisée du problème. De nombreux travaux détaillent la parallélisation précise de problèmes liés aux simulations et aux calculs scientifiques. Dans ces solutions il n'est pas rare de voir apparaître de nombreuses optimisations et parfois le mélange de différents modèles de parallélisation dans différentes parties du code, de façon à obtenir un programme encore plus performant. Ces codes sont souvent le résultat de plusieurs mois à plusieurs années de travail, et nécessitent parfois des collaborations ou des ressources humaines importantes. Ce type de parallélisation est très efficace mais nécessite une longue réflexion et une bonne connaissance des

architectures et des paradigmes parallèles. Tous les modèles et paradigmes de programmation parallèle peuvent être appliqués aux simulations numériques, des modèles induits par l'architecture mémoire jusqu'à la programmation d'architectures hybrides telles que les GPGPU, ou encore l'introduction d'instructions vectorielles dans les programmes. Toutefois, dans la plupart des cas, lorsque les ressources humaines, les collaborations ou le temps manquent, les scientifiques apprennent et utilisent un modèle de programmation parallèle facile à appréhender et à mettre en œuvre. Le livre de Bisseling [17] explique et illustre, par exemple, l'utilisation du modèle BSP et son implémentation en MPI pour paralléliser trois problèmes très importants du calcul scientifique : la décomposition LU, qui permet de décomposer une matrice comme un produit d'une matrice triangulaire inférieure L et d'une matrice triangulaire supérieure U, permettant de simplifier un système d'équations linéaires ; la transformée de Fourier, très utilisée en traitement du signal notamment ; et la multiplication d'une matrice creuse par un vecteur, qui est une opération très fréquente dans les calculs scientifiques. Dans ce cas, le modèle BSP est utilisé afin de résoudre des problèmes fréquents des mathématiques numériques, et une fois ces problèmes parallélisés, ils peuvent être utilisés dans des codes plus larges.

Une approche plus générale et plus facile de la parallélisation des simulations numériques est l'utilisation du modèle SPMD (Simple Program, Multiple Data). En effet, étant donné qu'une simulation scientifique est basée sur un maillage, il paraît plus simple de découper ce maillage en sous-parties, et que chacune soit gérée par un processeur différent. Dans ce cas, on procède à une *décomposition de domaine* (notion qui sera précisément décrite plus loin) et l'ensemble de la simulation est effectué par chaque processeur sur sa partie du domaine. Cependant, puisque les schémas numériques d'une simulation scientifique font intervenir un voisinage $N(x)$ pour les calculs de type *stencil*, il est nécessaire, dans ce type de programmes, de faire intervenir des communications entre les processeurs afin d'échanger les informations aux bords de leur sous-domaine. On définit un partitionnement du maillage μ en p parties $\mu_0, \mu_1, \ldots, \mu_{p-1}$, tel que $E_{\mu_i} \subset E_\mu$, $E_{\mu_i} \neq \emptyset$ pour tout i, et $E_{\mu_i} \cap E_{\mu_j} = \emptyset$ pour $i \neq j$. Le programme séquentiel de l'algorithme 1 devient alors l'algorithme 2. Le parallélisme de données est donc appliqué, dans ce cas, aux simulations numériques, et ce type de parallélisation offre des accélérations proches de l'idéal. Dans cette thèse, nous nous intéressons à la parallélisation SPMD de simulations numériques, pour des architectures à mémoire distribuée et hybride de type grappe, et nous cherchons à rendre ce type de parallélisation implicite.

2.3 Distribution de données

Depuis très longtemps, de nombreux travaux étudient les dépendances dans un calcul. Ces dépendances sont facilement identifiables à des graphes qui représentent, soit les étapes de calcul et leur dépendances (on parle alors de problèmes d'ordonnancement), soit les données du calcul et leur dépendances. Le cas le plus classique est de considérer un graphe $G = (V, E)$ où V représente l'ensemble des données du calcul et E l'ensemble des dépendances entre ces données. Le fort lien qui unit le calcul parallèle au partitionnement

Algorithme 2 : Algorithme parallèle de type SPMD d'une simulation basée sur un maillage.

```
Création du maillage μ
Partitionnement du maillage μ = {μ_0, μ_1, ... μ_{p-1}}
Création des quantités à simuler appliquées à μ
Initialisation des quantités et des paramètres
Définition du pas de temps, commun à tous les processeurs : Δt
Définition du temps maximal, commun à tous les processeurs : tmax
tant que t<tmax faire
    Communication de N(x) entre les processeurs
    pour chaque élément local ∈ E_{μ_i} de la bordure physique : b faire
        Calcul des conditions limites de la simulation
    fin
    pour chaque x ∈ E_{μ_i} local faire
        Obtention de σ(x, t−1) et σ(y, t−1), y ∈ N(x)
        Calcul de σ(x, t) en suivant le schéma numérique explicite
    fin
    t = t + Δt
fin
```

de graphe est donc évident. Le problème de partitionnement de graphe étant un problème NP-difficile [57], les méthodes mises en place pour le résoudre font appel à des heuristiques permettant d'approcher la solution exacte en un temps raisonnable. Nous allons aborder dans cette partie les modèles de partitionnement nécessaires à la compréhension de cette thèse, mais aussi le partitionnement de matrices et de maillages. Enfin, nous terminerons par la description de quelques partitionnements particuliers. L'ensemble des travaux présentés partent de l'hypothèse que tous les processeurs concernés par le partitionnement sont homogènes en matériel et donc en capaité de traitement (cluster). De plus, ces méthodes de partitionnement ne tiennent pas compte d'une topologie particulière du réseau.

2.3.1 Modèles de partitionnement

Dans cette section, nous allons décrire les différents modèles qui peuvent être utilisés pour représenter les dépendances d'un problème sous forme de graphe ou d'hypergraphe. Lorsque les dépendances du problème sont représentées correctement dans l'un de ces modèles, le problème de partitionnement posé est alors clair et peut être résolu à l'aide de solveurs existants, ou tout du moins à l'aide d'algorithmes et d'heuristiques connus. Nous ne développerons pas, dans cet état de l'art, les méthodes d'implémentation et les heuristiques existantes pour résoudre les problèmes de partitionnement. Ce sujet n'est, en effet, pas abordé dans cette thèse. Nous pouvons toutefois noter ici que la plupart des partitionneurs existants implémentent le partitionnement par la méthode *multilevel* qui est décrite dans de nombreux travaux [8, 49, 78, 123, 125]. Cette méthode consiste dans

un premier temps à rendre le graphe plus grossier en fusionnant certains sommets, puis à partitionner ce graphe dans un deuxième temps, et enfin à revenir petit à petit au graphe initial en raffinant le partitionnement. L'état de l'art des modèles de partitionnement que nous allons décrire ici s'inspire notamment du travail de Hendrickson et Al [66]. Nous ne décrirons toutefois ici que deux de ces modèles, qui seront abordés et utilisés dans cette thèse.

2.3.1.1 Partitionnement de graphe

Le partitionnement d'un graphe $G = (V, E)$ découpe l'ensemble des sommets V en p parties $V_0, \ldots, V_{p-1}$ telles que pour tout $i \in [\![0, p[\![$, $V_i \subset V$, $V_i \neq \emptyset$, et telles que pour tout $i, j \in [\![0, p[\![$, si $i \neq j$, alors $V_i \cap V_j = \emptyset$. Notons que tout sommet $v \in V$ peut être pondéré par $\omega(v)$, et chaque arête $e \in E$ peut, elle aussi, être associée à un poids, que l'on appellera plutôt un coût et que l'on note $c(e)$. Étant donné un sous-ensemble S de V, $\omega(S)$ est défini comme la somme des poids de chacun des nœuds de S. Le problème standard de partitionnement d'un graphe est alors d'effectuer ce partitionnement en respectant la contrainte d'équilibrage de charge suivante :

$$\max_{0 \leq i < p} \omega(V_i) \leq (1 + \epsilon) \frac{\omega(V)}{p}, \tag{2.15}$$

où $\epsilon > 0$ représente le pourcentage de déséquilibre toléré, et en minimisant le nombre d'arêtes coupées dans ce partitionnement. Cette dernière métrique, qui vise à couper le moins d'arêtes possible lors du partitionnement, est aussi appelée la métrique *edge-cut*. Le partitionnement standard de graphe est notamment implémenté dans les partitionneurs Chaco [67], METIS [77] et Scotch [100].

La méthode standard de partitionnement de graphe a longtemps été la seule méthode utilisée. Toutefois ses limites ont été très largement évoquées et résumées dans les travaux de Hendrickson et Al [66]. La critique de cette approche repose sur deux faiblesses : la métrique edge-cut et le modèle en lui-même. Nous n'allons pas évoquer ici l'ensemble des faiblesses de la métrique et de la méthode de partitionnement, nous pouvons toutefois noter deux points que nous considérons comme importants, et qui sont résolus par la méthode de partitionnement d'hypergraphe décrite par la suite.

La première faiblesse que nous souhaitons évoquer concerne la métrique edge-cut. Cette métrique dénombre les arêtes qui doivent être coupées suite au partitionnement mis en place. La limite de cette métrique vient du fait qu'elle n'est pas proportionnelle au volume de communication nécessaire dans un programme parallèle. En d'autres termes, cette métrique ne modélise pas correctement les communications pour la plupart des problèmes de partitionnement. Prenons un exemple afin d'illustrer cette caractéristique. Étant donné le graphe représenté dans la figure 2.9, partitionné en trois parties, une pour chaque processeur P_0, P_1 et P_2. Étant donné que chaque arête $e \in E$ représente un coût de communication $c(e) = 1 + 1 = 2$ (afin de représenter une communication symétrique), alors un partitionnement de graphe standard trouverait la métrique edge-cut comme égale à $c(e) \times 5 = 10$. Dans cet exemple, pourtant, nous pouvons observer que le sommet v_2 est relié par deux arêtes à la partition du processeur P_1, ce qui signifie qu'un unique

envoi de v_2 est nécessaire dans l'implémentation. En procédant ainsi pour les sommets v_5 et v_8 nous trouvons que le véritable volume de communication est égal à 7.

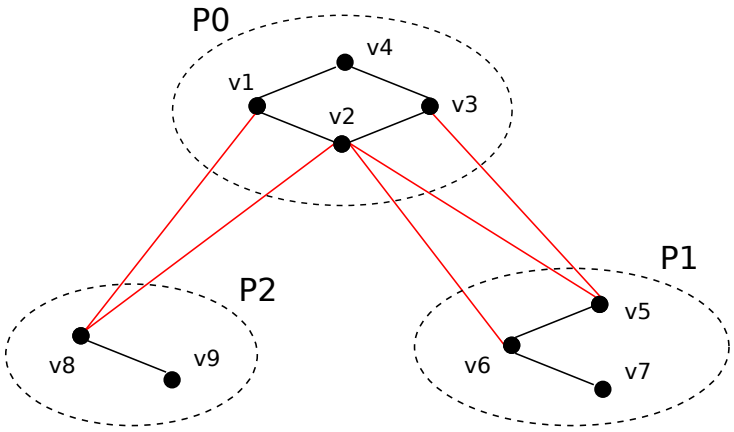


FIGURE 2.9 – *Graphe donnant un exemple de partitionnement où la métrique edge-cut ne représente pas le volume de communication.*

La deuxième faiblesse que nous évoquerons ici est le fait que la méthode standard de partitionnement de graphe ne permet d'exprimer que des dépendances symétriques. Une arête représente, en effet, un envoi de données des deux sommets la constituant. Ainsi la méthode de partitionnement manque d'expressivité pour certains problèmes asymétriques.

2.3.1.2 Partitionnement d'hypergraphes

Un *hypergraphe* $H = (V, \mathcal{N})$ est composé d'un ensemble de sommets, ou nœuds, noté V, et d'un ensemble $\mathcal{N}$ d'hyper-arêtes. Chaque hyper-arête est un sous-ensemble de V. Une hyper-arête est donc une généralisation de la notion d'arête dans un graphe, où plus de deux sommets de V peuvent être reliés entre eux. Dans le cas spécifique où chaque hyper-arête contient exactement deux sommets, on revient alors à la définition d'un graphe. Tout comme pour un graphe, tout sommet $v \in V$ d'un hypergraphe peut être pondéré par $\omega(v)$, et chaque hyper-arête $n \in \mathcal{N}$ peut, elle aussi, être associée à un poids, ou un coût, que l'on note $c(n)$. Ces poids sont généralement des réels positifs, mais dans cette thèse nous considérerons ces poids comme des entiers naturels. Étant donné un sous-ensemble S de V, $\omega(S)$ est défini comme la somme des poids de chacun des sommets de S.

Le partitionnement *p-way* d'un hypergraphe $H = (V, \mathcal{N})$ est défini par p sous-ensembles de V, $V_0, \ldots, V_{p-1}$ tels que pour tout $i \in [\![0, p[\![$, $V_i \subset V$, $V_i \neq \emptyset$, et tels que pour tout $i, j \in [\![0, p[\![$, si $i \neq j$, alors $V_i \cap V_j = \emptyset$. Le problème de partitionnement d'un hypergraphe est alors de trouver un partitionnement *p-way* qui satisfasse la

contrainte d'équilibrage (2.15), et qui minimise la métrique de coût suivante :

$$\sum_{n \in \mathcal{N}} c(n)(\lambda(n) - 1), \tag{2.16}$$

où $\lambda(n)$ est le nombre de parties connectées à une même hyper-arête $n \in \mathcal{N}$,

$$\lambda(n) = |\{V_i \ : \ 0 \leq i < p \text{ et } V_i \cap n \neq \emptyset\}|. \tag{2.17}$$

Cette métrique, que l'on cherche à minimiser, est appelée la *métrique-*$(\lambda - 1)$.

Le premier modèle de partitionnement d'hypergraphe est apparu dans les travaux de Çatalyürek et Al [26] . Son efficacité pour modéliser certains problèmes de partitionnement a été démontrée [28]. L'avantage principal de ce modèle est sa capacité à représenter exactement le volume de communications, ce qui n'est pas le cas en utilisant la métrique *edge-cut* du modèle de partitionnement de graphe. Reprenons, par exemple, le graphe $G = (V, E)$ de la figure 2.9, et construisons un hypergraphe $H = (V, \mathcal{N})$ où $|\mathcal{N}| = |V|$. L'ensemble des hyper-arêtes $\mathcal{N}$ est défini de façon à ce que chaque sommet $v_i \in V$ corresponde à une hyper-arête $h_i \in \mathcal{N}$ qui contient v_i et l'ensemble de ses sommets voisins dans G. Par exemple, l'hyper-arête du sommet v_2 contient alors les sommets v_2, v_8, v_6 et v_5. Cette hyper-arête contient donc des sommets des processeurs P_2 et P_1, son coût est donc de 2. Lors du partitionnement, on retrouve alors la métrique de coût de communication définie dans l'équation (2.16) qui est bien égale à 7 pour cet exemple. Pour finir, la méthode de partitionnement d'hypergraphe permet la représentation de problèmes asymétriques.

2.3.2 Cas particulier du partitionnement de matrices

Le modèle de partitionnement d'hypergraphes a été utilisé dans de nombreux travaux afin de représenter les communications d'une multiplication de matrice creuse par un vecteur [28]. Ce traitement est l'un des plus courants dans les calculs scientifiques et a été très largement étudié. Un ensemble de méthodes de partitionnement, spécifiques à ce problème, a été élaboré. Un parallèle pouvant être fait de plusieurs façons entre un hypergraphe, ou un graphe, et une matrice creuse, les méthodes et les partitionneurs qui ont été développés pour ce type de traitements permettent également de résoudre d'autres problèmes de partitionnement. Dans cette thèse, le partitionneur Mondriaan [120], qui implémente plusieurs de ces techniques, est utilisé. Nous allons donc décrire, dans cette section, l'ensemble des modèles de partitionnement qui ont été mis en place pour le problème de multiplication *matrice creuse-vecteur*. Nous ne décrirons pas, en revanche, comment utiliser ces modèles sur la multiplication *matrice creuse-vecteur* en elle-même, puisque cette thèse ne s'intéresse pas particulièrement à ce problème. Ces détails peuvent être trouvés dans les travaux de Bisseling et Al [18].

2.3.2.1 Partitionnement à une dimension

Voyons une première façon de transformer une matrice vers un hypergraphe. Considérons une matrice creuse A de taille $m \times n$. Notons alors $a_{i,j}$ ses coefficients avec $i \in [\![1, m[\![$

et $j \in [\![1, n[\![$. On peut alors considérer que chaque colonne j de la matrice A est représentée par un sommet de l'hypergraphe avec un poids $\omega(j)$ égal au nombre de valeurs non nulles dans la colonne j. Considérons ensuite que chaque ligne i de la matrice A est représentée par une hyper-arête qui contient les sommets j pour lesquels $a_{i,j} \neq 0$. Pour finir, considérons que le coût d'une hyper-arête est égal à 1. Cette représentation d'une matrice creuse A par un hypergraphe H_r est appelée le modèle *row-net*, qui signifie que les hyper-arêtes (net) représentent les lignes de la matrice (row). Dans ce cas un partitionnement *p-way* de l'hypergraphe H_r amène à un partitionnement à une dimension, ou 1D, de la matrice A. Ce partitionnement distribue donc les colonnes de la matrice A en p parties différentes. Chaque colonne étant pondérée par ω, le nombre de valeurs non nulles dans la colonne, le partitionnement de H_r distribue de façon équilibrée les valeurs non nulles de la matrice A en suivant la contrainte d'équilibrage définie dans l'équation (2.15). Pour finir, le volume de communications causé par la séparation d'une ligne de A dans plusieurs parties est minimisé par le fait qu'une ligne représente une hyper-arête et minimise donc la métrique-$(\lambda - 1)$ représentée dans l'équation (2.16). Notons, pour terminer, qu'il est également possible de faire un partitionnement 1D de la matrice A par le modèle *column-net* qui, à l'inverse, associe les lignes de A aux sommets de l'hypergraphe H_c et les colonnes de A aux hyper-arêtes de H_c.

2.3.2.2 Partitionnements à deux dimensions

Un partitionnement à deux dimensions, ou 2D, de la matrice A est également possible en procédant de plusieurs façons que nous allons décrire ici. Dans un partitionnement de matrice 2D, les colonnes comme les lignes de la matrice peuvent être découpées en plusieurs parties, ce qui implique des communications dans les deux directions. Le partitionnement 2D d'une matrice creuse a l'avantage de généraliser le problème, ce qui peut conduire à une solution plus intéressante avec un meilleur équilibrage ou moins de communications. Nous évoquerons ici quatre modèles principaux de partitionnement à deux dimensions.

Méthode coarse-grain. Pour partitionner une matrice A, la méthode coarse-grain a la particularité d'essayer de conserver les rapprochements naturels des valeurs non nulles de la matrice par colonnes et par lignes, tout comme le fait un partitionnement 1D, mais en prenant en compte les deux dimensions. Il existe plusieurs types de méthodes dites coarse-grain. On peut, par exemple, noter le partitionnement cartésien, très utilisé pour le partitionnement de maillages cartésiens (nous reviendrons plus en détails dessus). Nous pouvons également évoquer l'approche Mondriaan [120] qui consiste à successivement partitionner en deux (ou bipartitionner) la matrice jusqu'à atteindre p parties. A chaque bipartitionnement, les méthodes row-net et column-net sont essayées, et celle proposant le meilleur partitionnement est conservée. Les deux hypergraphes H_r et H_c sont donc partitionnés à chaque itération. Ainsi, cette méthode effectue des partitionnements 1D mais qui peuvent être effectués dans les deux directions ce qui permet d'obtenir un partitionnement 2D et un plus grand nombre de solutions potentielles.

Méthode fine-grain. À l'inverse de la méthode coarse-grain, qui se base sur l'unité des lignes et des colonnes de la matrice, la méthode fine-grain [30] se propose de partitionner chaque valeur non-nulle de façon indépendante dans p partitions. Pour cela un nouveau type d'hypergraphe, noté H_f, est construit à partir de la matrice A. Chaque sommet de cet hypergraphe représente non plus les lignes ou les colonnes de la matrice, mais chaque élément non nul de la matrice. Deux types d'hyper-arêtes sont alors représentées. Une hyper-arête ligne i contient l'ensemble des sommets correspondants aux valeurs non nulles de la ligne i de A. Une hyper-arête colonne j, quant à elle, contient l'ensemble des sommets correspondants aux valeurs non nulles de la colonne j de A. Le nombre total de sommets dans l'hypergraphe H_f est égal au nombre de valeurs non nulles, et le nombre d'hyper-arêtes est au plus égal à $m + n$. Une fois cet hypergraphe construit, il peut être partitionné en p sous-ensembles de sommets en suivant la contrainte d'équilibrage (2.15) et en minimisant la métrique de coût (2.16).

Méthode hybride. La méthode hybride [18] reprend le principe de l'approche Mondriaan par bipartitionnements successifs, mais en ajoutant aux partitionnements des hypergraphes H_r et H_c le partitionnement de l'hypergraphe H_f de la méthode fine-grain.

Méthode medium-grain. La méthode medium-grain a récemment été proposée dans les travaux de Pelt et Al [101]. Cette méthode sépare tout d'abord la matrice A en deux matrices A^c et A^r. La valeur $a_{i,j}$ est ainsi assignée à la matrice A^r si le nombre de valeurs non nulles dans la ligne i est plus grand que dans la colonne j, et à A^c dans le cas contraire. La méthode crée alors une matrice :

$$B = \begin{bmatrix} I_n & (A^r)^T \\ A^c & I_m \end{bmatrix}, \tag{2.18}$$

où I_m est la matrice identité de taille $m \times m$, et I_n la matrice identité de taille $n \times n$. La méthode de partitionnement 1D row-net est ensuite utilisée pour partitionner la matrice B. La matrice A^r étant transposée dans B, il s'agit là encore d'un partitionnement à deux dimensions, où la dimension de partitionnement, pour chaque valeur, est choisie en fonction de son attribution dans A^r ou A^c.

2.3.3 Cas particulier du partitionnement de maillages

Cette thèse traite de solutions de parallélisme implicite pour le cas des simulations scientifiques dont la résolution est basée sur des maillages. Nous allons, dans cette partie, étudier le cas spécifique, et pratique, du partitionnement de maillages pour les applications parallèles. Nous allons, tout d'abord étudier l'état de l'art pour le cas des maillages réguliers à deux dimensions, puis nous traiterons le cas des maillages non-structurés.

2.3.3.1 Maillages à deux-dimensions réguliers

Comme nous l'avons vu, un maillage régulier à deux dimensions peut être de deux types. Soit un maillage cartésien, soit un maillage curvilinéaire, mais qui peut alors être

ramené à un maillage cartésien. Un maillage cartésien est bien souvent représenté, dans les esprits, comme une matrice. Par conséquent un partitionnement fine-grain pourrait, par exemple, être envisagé pour partitionner les éléments non nuls de la matrice (tous les éléments dans le cas d'une matrice dense). Un maillage cartésien étant dense, des méthodes plus directes, et plus simples à mettre en œuvre, permettent d'obtenir rapidement des partitionnements équilibrés et contenant peu de communications.

Notons par exemple le partitionnement *rectiligne* [97], qui est obtenu en partitionnant tout d'abord les lignes du maillage en P parties, puis les colonnes en Q parties, tel que $p = PQ$. On peut ensuite assigner chaque combinaison obtenue à chaque processeur. Une variante de ce partitionnement est d'utiliser la même technique mais suivant une unique dimension. On pourra aussi appeler le partitionnement rectiligne 2D, le partitionnement par *blocs*, et le partitionnement rectiligne 1D, le partitionnement par *blocs-lignes*.

Dans certains cas, un maillage cartésien peut être non-uniforme. C'est le cas des maillages dits *adaptatifs*. Ce type de maillage peut être intéressant pour résoudre des EDP, puisqu'il permet d'adapter le maillage avec plus ou moins de précision (de points) suivant les zones d'intérêt du domaine. Dans ce cas, il peut être à la fois plus compliqué d'équilibrer le partitionnement, mais aussi de minimiser les communications. Les travaux de Berger et Al [13] ont introduit en 1987 le partitionnement par bissection récursive orthogonale (ORB) pour ce type de maillages. La figure 2.10 représente les trois partitionnements de maillages 2D introduits ici.

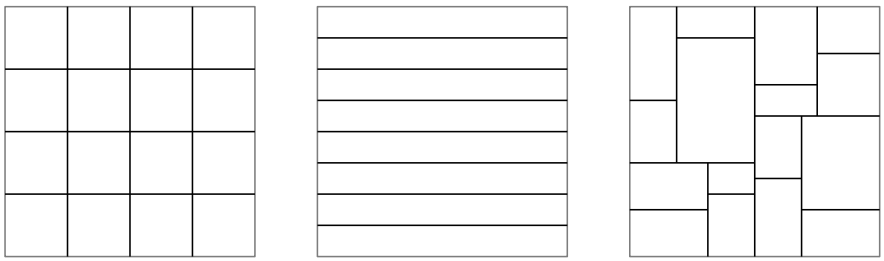


FIGURE 2.10 – *De gauche à droite : partitionnement en blocs, en blocs-lignes et bissection récursive orthogonale*

2.3.3.2 Maillages non-structurés

Nous venons de voir que le partitionnement des maillages réguliers est un cas de partitionnement relativement simple et pour lequel un grand nombre de possibilités de résolution est disponible. Certains maillages sont eux beaucoup plus compliqués à partitionner de par leur structure irrégulière. La méthode des éléments finis, que nous avons décrite dans la partie 2.2.4.3, mène dans la plupart des cas à la création d'un maillage non-structuré qui permet de représenter avec fidélité et avec plus ou moins de précision la surface ou le volume d'un objet. La plupart du temps les cellules de ces maillages représentent des triangles (2D) ou des tétraèdres (3D), ce qui permet, suivant la taille des mailles, de pouvoir représenter très précisément les surfaces ou les volumes. Une maille

peut avoir une taille quelconque et peut être un triangle de forme quelconque dans l'espace. Le voisinage y est donc régulier, dans le sens où toutes les cellules ont par exemple trois cellules voisines (dans le cas de triangles), mais les structures de données permettant de représenter un maillage non structuré sont elles plus complexes et plus lourdes que dans un maillage cartésien. Par conséquent, là où un maillage cartésien peut facilement être identifié à une matrice, le maillage non-structuré est lui plus facilement représenté par un graphe. Le problème de partitionnement d'un maillage non-structuré repose donc sur le fait de trouver la bonne représentation du maillage en graphe pour ensuite pouvoir le partitionner. Quatre représentations sont très souvent utilisées dans la littérature :

— La première, et la plus simple, est de considérer chaque point du maillage comme un sommet d'un graphe, et chaque arête d'une face du maillage comme une arête du graphe. Cette représentation est appelée le graphe *nodal* du maillage [122].
— La deuxième représentation, appelée le graphe *dual* du maillage [46, 106], associe chaque cellule du maillage à un sommet du graphe. Deux sommets du graphe sont reliés par une arête si deux cellules du maillage ont un côté, ou une face, en commun.
— La troisième représentation combine le graphe nodal et le graphe dual afin d'obtenir une représentation plus précise sur le maillage [122].
— Enfin, le graphe *dual-diagonal* représente chaque cellule du maillage par un sommet, et deux sommets sont reliés par une arête si les cellules ont un point en commun dans le maillage. Notons que cette représentation peut elle aussi être combinée au graphe dual pour représenter avec plus de précision le maillage.

La figure 2.11 illustre le graphe nodal, le graphe dual et le graphe dual-diagonal d'un maillage non structuré 2D. Une fois que la représentation du maillage par un graphe

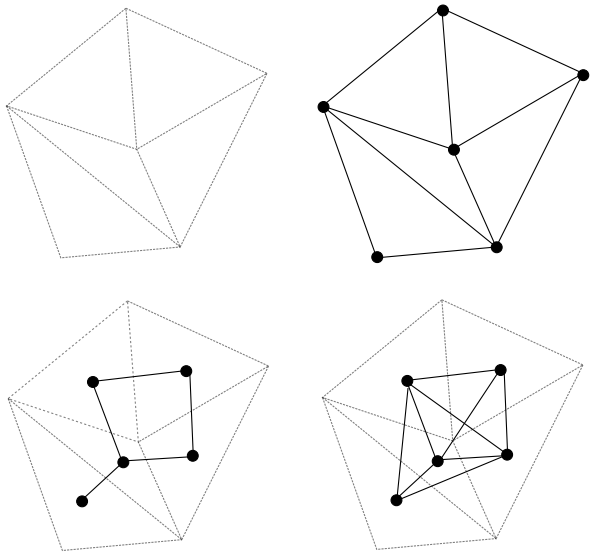


FIGURE 2.11 – *De gauche à droite et de haut en bas : le maillage non structuré 2D, son graphe nodal, son graphe dual et son graphe dual-diagonal.*

est choisie, les partitionneurs de graphes peuvent être utilisés, comme par exemple Jostle [122], Metis [46, 106] et Scotch [100, 106].

Dans les travaux de Zhou et Al [131], le partitionnement d'hypergraphe est utilisé pour partitionner un maillage non-structuré 3D contenant 1.07 milliard de cellules, 163840 processeurs. Le maillage y est représenté par un hypergraphe dans lequel chaque sommet est associé à une cellule du maillage, et chaque hyper-arête correspond à une cellule et aux cellules partageant une face avec celle-ci. Le partitionneur Zoltan [27] est ensuite utilisé. Il est donc également possible d'utiliser le modèle de partitionnement d'hypergraphes pour partitionner un maillage non-structuré et représenter plus fidèlement le volume de communications.

2.3.4 Partitionnements particuliers

2.3.4.1 Méthodes à contraintes et objectifs multiples

Considérons un problème de partitionnement d'hypergraphe, ou de graphe, défini comme dans les parties précédentes. On peut alors appeler la contrainte d'équilibrage de l'équation (2.15) la *contrainte* du partitionnement, et la minimisation des métriques $(\lambda - 1)$ de l'équation (2.16), ou edge-cut, l'*objectif* du partitionnement. Un partitionnement à contraintes multiples [9, 108] consiste alors à appliquer un tableau de poids à chaque sommet de l'hypergraphe ou du graphe au lieu d'un simple poids. Ce tableau représente les multiples contraintes d'équilibrage à respecter lors du partitionnement. De même un partitionnement à objectifs multiples [58] appliquera un ensemble de coûts de communications à une hyper-arête de l'hypergraphe, ou à une arête du graphe.

Pour effectuer un partitionnement à contraintes multiples, Aykanat et Al [108] ont modifié l'algorithme multilevel dans ses trois phases. En effet, les phases de réduction, de bi-partitionnement et de raffinement ont été modifiées pour tenir compte de plusieurs contraintes de partitionnement. Dans les travaux de Schloegel et Al [58], l'algorithme de partitionnement pour objectifs multiples s'effectue en trois phases. Tout d'abord un algorithme de partitionnement *k-way* du partitionneur METIS [77] est appliqué pour chacun des objectifs séparément. Un nouveau poids est ensuite attribué à chaque arête du graphe initial. Ce poids est calculé comme une fonction des différents poids de l'arête (objectifs multiples), du meilleur résultat de la métrique edge-cut obtenu dans la première phase, et du vecteur de préférence des objectifs précisé par l'utilisateur. Enfin, un dernier partitionnement k-way est opéré sur le graphe avec les nouvelles pondérations d'arêtes.

2.3.4.2 Méthode pour les calculs à phases

Certains calculs scientifiques ou simulations ont la particularité d'être organisés en plusieurs phases. Cette organisation peut être de plusieurs types dans une simulation. Tout d'abord, il est possible qu'il s'agisse de différentes phases de calcul, mais toutes exécutées sur l'ensemble du maillage. Dans ce cas, les différentes phases peuvent avoir un impact sur le type de communications et il est alors possible d'utiliser des partitionnements à contraintes ou objectifs multiples. Il est également possible que les différentes

phases du calcul soient exécutées sur des maillages différents, soit complètement distincts les uns des autres, soit reliés entre eux par une composition de maillages (section 2.2.2.2). Nous traitons plus en détails ce cas dans la section 5.5 de cette thèse. Enfin, il est également possible que les différentes phases d'un calcul agissent sur différentes parties d'un même maillage. Dans ce cas une contrainte d'ordonnancement apparaît à la fois sur les calculs, mais aussi sur le maillage. Les travaux de Walshaw et Al [125] traitent de ce type de calculs à phases. Dans ces travaux de partitionnement, les sommets du graphe vont être classifiés afin de déterminer la phase qui les concerne. Le premier sous-ensemble de sommets est alors partitionné, puis les sous-ensembles suivants seront partitionnés à leur tour en tenant compte des partitionnements précédents, grâce à la notion de point stationnaire introduite dans ces travaux. Notons que la méthode est également capable de traiter des sommets qui appartiennent à plusieurs phases du calcul.

Le problème de partitionnement de graphes est un problème qui a largement été étudié et dont quelques représentations ont étés présentées dans cette section. Nous utiliserons, dans le chapitre 5 plus spécifiquement, certaines de ces notions afin de présenter le problème de partitionnement de réseaux, et deux méthodes de résolution.

2.4 Le parallélisme implicite

Nous allons désormais aborder le cœur de cette thèse : le parallélisme implicite. Derrière ce terme se cache le fait de vouloir apporter un accès facile, simplifié, voire transparent, au calcul haute performance et au parallélisme à des utilisateurs non spécialistes, et même non-informaticiens. En effet, si la plupart des scientifiques ont de plus en plus besoin des machines parallèles pour obtenir des simulations intéressantes, ils ne savent pas pour autant les utiliser à leur pleine capacité, soit par manque de temps et de ressources humaines, soit par manque de connaissances sur les architectures matérielles utilisées. De nombreux travaux sur le parallélisme implicite ont vu le jour presque simultanément avec l'arrivée d'architectures parallèles, complexes à programmer. Ce domaine de recherche est très actif et le sera probablement de plus en plus étant donné la complexité des architectures parallèles actuelles et à venir (hiérarchie de mémoires complexes, systèmes massivement multi-cœurs, systèmes hybrides etc.). Nous allons présenter, dans cet état de l'art, un aperçu des solutions les plus utilisées et les plus reconnues du parallélisme implicite. Pour cela, nous allons tout d'abord présenter des solutions permettant de classifier les problèmes parallèles, nous évoquerons ensuite quelques-uns des nombreux langages et des nombreuses bibliothèques de parallélisme partiellement implicites. Enfin nous entrerons à proprement parlé dans les solutions de parallélisme implicite totale, que nous essaierons de classer par niveau d'abstraction, le niveau d'abstraction le plus haut (qui ne correspond pas nécessairement au meilleur niveau d'abstraction, et c'est l'une des discussions de cette thèse) étant celui qui cache le plus de technicités et qui demande le moins d'apprentissage à l'utilisateur. Ainsi nous évoquerons les bibliothèques de parallélisme implicite générales, les solutions à patrons, puis pour terminer les solutions spécifiques au domaine du calcul scientifique.

2.4.1 Classification de problèmes et aide à la parallélisation

Le niveau d'abstraction le plus bas du parallélisme est de considérer qu'il est préférable de laisser les utilisateurs coder leurs propres programmes parallèles, mais de les aider dans la conception de ces programmes. Les modèles de programmation tels que BSP [92,119] ou MPI [61], ainsi que le paradigme SPMD, décrits précédemment, sont des exemples de solutions permettant de simplifier le parallélisme et proposent un niveau d'abstraction plus haut que la programmation parallèle de base. En effet, pour certains types d'architectures parallèles et pour certains types de problèmes, ces modèles de programmation parallèle peuvent être utilisés relativement facilement. Il est d'ailleurs fréquent d'initier les scientifiques à la programmation parallèle par ce type de modèles [17,87].

Toutefois, lorsque les algorithmes deviennent plus compliqués à mettre en œuvre, comme par exemple dans le cas de problèmes irréguliers, les modèles les plus simples sont souvent limités, et une réflexion différente sur la conception du programme parallèle est souvent nécessaire. Les travaux de Pingali et Al [103] proposent une classification des algorithmes afin de mieux identifier la façon dont ils peuvent être parallélisés efficacement. Il a ainsi été proposé *The TAO of Parallelism in Algorithms*, qui peut être vu comme une abstraction des algorithmes. Cette abstraction permet, d'une part, d'extraire les propriétés importantes pour la parallélisation du problème et, d'autre part, de mettre de côté les propriétés n'entrant pas en compte dans les choix de parallélisation. Dans l'*analyse-TAO*, la définition d'un algorithme est inspirée de l'aphorisme de Niklaus Wirth [129] : *Program = Algorithm + Data structure*. L'abstraction proposée par l'analyse-TAO est appelée *operator formulation of algorithms*. L'algorithme y est traduit comme un graphe représentant les opérations effectuées sur des types de données abstraits, noté *graphe ADT*. L'analyse-TAO se décompose en trois étapes. Tout d'abord la définition de la *topologie*, qui représente la structure de données sur laquelle les calculs sont effectués, puis les *nœuds actifs*, qui représentent les éléments à calculer dans une opération donnée, et enfin les *opérateurs*, qui représentent les actions à effectuer sur les nœuds actifs.

Une simulation scientifique pour laquelle des schémas numériques explicites sont appliqués sur un maillage fixe (à l'inverse d'un maillage adaptatif) est définie comme suit, dans l'analyse-TAO :

- Topologie : La topologie nécessaire dépend du type de maillage utilisé. Elle peut être structurée, comme par exemple pour des maillages cartésiens, ou non-structurée, pour les maillages du même nom.
- Nœuds actifs : Dans l'analyse-TAO, l'algorithme d'une simulation scientifique est appelé *topology-driven*. Cela signifie que l'ensemble des éléments du maillage sont calculés à chaque itération de temps. De plus, si les schémas numériques à calculer sont explicites, leurs calculs ne dépendent que de l'itération précédente et les éléments du maillage peuvent donc être calculés de façon *non-ordonnée*.
- Opérateurs : Les opérations mises en place dans une simulation scientifique représentent les calculs des schémas numériques. Dans le cas d'un maillage non adaptatif, ou fixé, la morphologie du maillage n'est pas modifiée au cours de l'algorithme. L'analyse-TAO appelle ce type de simulation un calcul local (*local computation*).

Notons que dans le cas d'une simulation utilisant des schémas numériques implicites, l'algorithme est appelé *data-driven*, ce qui signifie que les calculs de certains éléments du maillage peuvent rendre calculables d'autres éléments. Enfin, dans le cas d'un maillage adaptatif, les opérateurs sont de type *morph*, ce qui signifie que le maillage peut être modifié à chaque itération de temps de la simulation.

Ce premier niveau d'abstraction vise à simplifier la classification des algorithmes parallèles et permet donc d'obtenir une première approche simplifiée du parallélisme et du calcul haute performance. Toutefois, il ne résout pas le fait que l'utilisateur ne connaît pas suffisamment les architectures et les détails techniques de la programmation parallèle pour écrire des programmes parallèles. Ces modèles de programmation et ces classifications sont, en revanche, très utilisés afin d'élaborer des solutions de parallélisme implicite d'un niveau d'abstraction plus élevé.

2.4.2 Solutions partiellement implicites

Il existe un très grand nombre de bibliothèques et de langages permettant de simplifier l'utilisation des machines parallèles, et donc d'écrire des programmes pour ces machines. Toutefois, ces solutions proposent un parallélisme qui n'est que partiellement implicite, laissant à l'utilisateur la charge de quelques notions parallèles, qui peuvent paraître, pour certaines, simples sur de petits programmes, mais qui peuvent s'avérer très compliquées pour des calculs eux-mêmes complexes. Les niveaux de parallélisme implicite proposés sont très variés et nous allons évoquer ici quelques unes de ces solutions.

À un niveau d'abstraction relativement bas, on peut tout d'abord noter les langages ou interfaces de programmations à base de directives. Citons deux de ces solutions, *High Performance Fortran* [93] (HPF) et OpenMP [34]. Les langages de directives, et particulièrement HPF ont été largement critiqués, et notamment accusés de rejeter certains détails du parallélisme, potentiellement très techniques, sur l'utilisateur. En effet, HPF demande, par exemple, à l'utilisateur de préciser les alignements des données entre elles, ce qui garantit leur placement sur un même processeur. De même, HPF demande à l'utilisateur de préciser des directives de distribution de données (en bloc, ou de façon cyclique élément par élément, par exemple). OpenMP, de son côté, a réussi à devenir une référence pour paralléliser facilement, mais pas nécessairement efficacement, des applications sur architectures à mémoire partagée. Rappelons que si OpenMP est initialement un modèle de programmation induit par et fait pour les architectures à mémoire partagée, son implémentation peut être effectuée sur différents modèles d'exécution et notamment pour des architectures à mémoire distribuée, en utilisant, par exemple, une architecture *DSM* [79]. Bien que ses performances soient limitées, OpenMP est très utilisé dans les calculs scientifiques. Les directives OpenMP sont, en effet, peu nombreuses et moins techniques que celles proposées par HPF. Deux types de parallélisation sont possibles en utilisant OpenMP. La plus simple, et la plus connue des scientifiques, est la méthode dite *fine-grain*, ou à grain fin. Elle consiste en la parallélisation automatique de boucles *for*. L'unique difficulté pour l'utilisateur est alors de détecter quelles variables sont locales à la boucle et quelles variables doivent être partagées par les différents processus

créés automatiquement. Toutefois, il n'est pas rare que cette parallélisation de boucle, très limitée, ne soit pas suffisante pour obtenir des performances intéressantes, et même parfois acceptables, dans les calculs scientifiques. Le deuxième type de programmation OpenMP est alors utilisé et s'appelle *coarse-grain*, ou à gros grain. Dans ce cas, on peut noter deux types de solutions. Dans la première l'utilisateur a la charge de définir une zone de code parallèle, en utilisant la directive *#pragma omp parallel*, où l'ensemble des processus, créés automatiquement, exécuteront la même portion de code. L'utilisateur devra également définir les variables locales et partagées, et on retrouve dans ce type de parallélisation des problèmes de décomposition de domaine. Cette solution coarse-grain peut donc s'apparenter au parallélisme de données. Le deuxième type de programmation coarse-grain qui peut être envisagé est la définition par l'utilisateur de sections pouvant directement être assignées à des processus différents. La directive est alors *#pragma omp parallel sections*. Dans ce cas l'utilisateur doit identifier des tâches pouvant être effectuées en parallèle par plusieurs processus, cette solution s'apparente donc au parallélisme de tâches. L'utilisation des méthodes coarse-grain, permet souvent d'obtenir de meilleures performances, toutefois le niveau d'abstraction proposé est plus bas, et le parallélisme peu implicite, tout comme dans l'utilisation de HPF.

Les langages BSPLib [69] et Co-array Fortan [98] peuvent ensuite être évoqués. Là encore, il ne s'agit pas d'un parallélisme implicite total, toutefois la programmation parallèle y est simplifiée par un nombre de concepts limités et par l'utilisation d'un modèle de programmation proche de BSP. Ces langages permettent notamment de simplifier le paradigme à passage de messages, proposé par exemple par MPI [61]. BSPLib, par exemple, ne demande à l'utilisateur que d'expliciter les envois de données et les synchronisations. Le reste du travail est effectué directement par la bibliothèque grâce au modèle de programmation BSP.

ZPL [32] est un langage parallèle basé sur la définition et l'utilisation de tableaux. Il est, pour cette raison, notamment très adapté aux calculs matriciels. La distribution des tableaux est effectuée automatiquement lors de l'exécution, et les programmes implémentés suivent le paradigme SPMD. Si HPF, Co-array, BSPLib et ZPL, par exemple, sont des langages proches du paradigme SPMD, ce qui signifie que chaque thread exécute le même code sur des données différentes, les langages X10 [35] et Chapel [31] proposent, quant à eux, un modèle plus général permettant de contrôler un ensemble d'opérations concurrentes. Notons enfin que les langages Co-array et X10, parmi d'autres langages, utilisent le modèle de programmation parallèle PGAS (*Partitioned Global Adress Space*) [6], déjà évoqué dans la partie 2.1.2.2.

2.4.3 Solutions générales de parallélisme implicite

Le niveau d'abstraction suivant permet, lui, de cacher de façon beaucoup plus prononcée le parallélisme aux utilisateurs. Nous évoquons ici les solutions de parallélisme implicite, que l'on qualifie de générales, à l'inverse des solutions spécifiques que nous évoquerons par la suite. Il s'agit à proprement parlé, de solutions de parallélisme implicite totales. Afin de comprendre cette classe de solutions, nous allons prendre l'exemple de la *librairie standard template* [95] du C++ (STL). Cette bibliothèque, très générale, per-

met d'instancier et d'utiliser des conteneurs, comme par exemple des vecteurs, des listes, des dictionnaires etc., et même de les imbriquer entre eux. Un ensemble d'algorithmes peuvent ensuite être appliqués sur ces conteneurs, mais il est également possible d'écrire ses propres programmes au moyen d'un outil principal : l'itérateur. L'itérateur permet, en effet, de se déplacer dans un conteneur, mais aussi d'accéder aux valeurs qui y sont associées. La STL permet d'écrire de façon simplifiée des programmes en C++. En un sens donc, la STL est une solution de "conteneurs implicites", qui permet de cacher la complexité de gestion de conteneurs en C++ et de faciliter leur utilisation. Nous évoquons, par le terme bibliothèques de parallélisme implicite générales, des bibliothèques équivalentes à la STL mais qui permettent d'écrire des programmes parallèles. Il s'agit donc de solutions permettant de mettre en place le paradigme de parallélisme de données, ou SPMD, par la parallélisation des conteneurs et de leur manipulation. Dans ce cas, les programmes écrits sont très généraux et touchent potentiellement un très grand nombre de domaines scientifiques. Certaines de ces bibliothèques sont plus spécifiquement décrites ici.

Bibliothèques sur conteneurs généraux. STAPL [25] signifie *Standard Template Adaptative Parallel Library* et il s'agit d'une version parallèle de la STL, que nous venons de décrire. Cette bibliothèque emprunte donc un certain nombre de concepts de la STL, et son but est d'offrir autant, ou plus, de possibilités de codage que la STL, tout en produisant des programmes parallèles. STAPL est composée de deux composants principaux, le premier est le concept de *pContainer* qui représente une structure de données distribuée représentant un conteneur distribué (tableau [113], liste [115], etc.). Le deuxième concept, *pAlgorithm*, représente, quant à lui, un algorithme à appliquer sur un pContainer. Il est possible dans STAPL, comme dans la STL, d'imbriquer des pContainers et donc d'imbriquer également des appels à des pAlgorithms. Le niveau d'abstraction proposé par STAPL est illustré par le concept de *pView* [24], qui représente une généralisation du concept d'itérateur. Le concept pView permet le parallélisme via un accès, d'ordre inconnu, à l'ensemble des éléments d'un conteneur. Enfin, le *STAPL parallel container framework* [114] permet d'écrire de nouveaux pContainers de façon simplifiée. Notons également la bibliothèque PSTL [64], dont les buts sont proches de STAPL. PSTL est une version parallèle de la STL, mais cette bibliothèque cherche à rester compatible avec la STL là où STAPL propose de nouveaux conteneurs qui n'existent pas dans la STL comme les matrices (*pMatrix*) et les graphes (*pGraph*). La bibliothèque *Threading Building Blocks* (TBB) [105] implémente, elle aussi, certains concepts équivalents à la bibliothèque STAPL mais ne vise initialement que les architectures à mémoire partagée (bien qu'une implémentation utilisant une architecture DSM puisse là encore être envisagée). STAPL, de son côté, fonctionne de base à la fois pour les architectures à mémoire partagée et distribuée. Enfin, Thrust [70] est une bibliothèque proposant elle aussi un équivalent de la STL en parallèle pour des architectures GPU et hybrides CPU-GPU.

Bibliothèques sur graphes. Si les bibliothèques STAPL, PSTL et TBB se veulent généralistes pour tout type de conteneurs, certaines bibliothèques, elles aussi basées sur

des conteneurs et des algorithmes, se concentrent sur les graphes, ce qui représente un problème difficile à gérer en lui-même. *Parallel Boost Graph Library* (PBGL) [62] est une bibliothèque parallèle générale sur les graphes. PBGL est une version parallélisée de BGL (*Boost Graph Library*) [1], et reste entièrement compatible avec cette version séquentielle. BGL et PBGL sont des bibliothèques implémentées dans l'ensemble de bibliothèques Boost [3]. Elles en héritent donc les forts concepts de généricité et d'efficacité. BGL, et donc PBGL, sont des bibliothèques C++ génériques visant à pouvoir exprimer un maximum de problèmes, tout en proposant une implémentation efficace. Leur implémentation est, pour cela, basée sur les concepts avancés de méta-programmation [3] et de spécialisation de template [5]. Il en résulte, pour des scientifiques non-informaticiens, une programmation techniquement difficile à comprendre, d'autant plus que certains paramètres de spécialisation, permettant de rendre la solution plus efficace, sont loin des préoccupations des scientifiques, comme par exemple des informations sur le type de représentation du graphe souhaité, ou sur la distribution à effectuer pour le parallélisme. En souhaitant être une bibliothèque générale à tous les problèmes de graphes, elle restreint son utilisation à des utilisateurs avancés du C++ (bien qu'aucun code parallèle ne soit demandé à l'utilisateur). La bibliothèque CGMGraph [33] implémente, tout comme PBGL, un certain nombre d'algorithmes sur les graphes, comme par exemple les composantes connexes, les arbres couvrants etc. Toutefois, le paradigme de programmation des deux méthodes est différent. CGMGraph est une bibliothèque orientée objet alors que PBGL est orientée programmation générique.

Il peut sembler que ce niveau d'abstraction est idéal. En effet, étant très général, il touche l'intégralité du monde scientifique et permet d'écrire des programmes parallèles en cachant potentiellement intégralement les détails du parallélisme. Cependant, de par la généralité de ces solutions, il est difficile de proposer des optimisations spécifiques à un domaine. Ces solutions peuvent être performantes mais ne peuvent être à la hauteur d'un parallélisme manuel et optimisé pour un problème spécifique de simulation scientifique. De plus, le souhait de généricité de ces solutions, et leurs paramétrages parfois complexes, peut être à l'origine de nouvelles difficultés pour l'utilisateur. Une généricité trop faible, à l'inverse, peut nuire aux performances de la solution.

2.4.4 Solutions à patrons

Les bibliothèques de parallélisme implicite générales, comme nous venons de le voir, cherchent à cacher le parallélisme par le biais de conteneurs, d'algorithmes et d'itérateurs (tableaux 1D, 2D, graphes etc.). Nous allons maintenant aborder des solutions proposant un niveau d'absrtaction que nous considérons plus haut puisque davantage de détails sont cachés à l'utilisateur. Ces solutions sont, elles aussi, basées sur des structures de données implicitement distribuées, mais proposent, de plus, d'identifier les opérations effectuées dans un programme comme un ensemble de patrons de programmation (ou *patterns*). Ces patterns seront ensuite responsables de la parallélisation implicite des opérations séquentielles du code de l'utilisateur. Les patrons proposent un haut niveau d'abstraction. Par exemple, ce type de solutions cache généralement la navigation dans

les structures de données, la notion d'itérateur n'est alors plus nécessaire. Nous allons décrire ici deux grandes familles de solutions à patrons. Tout d'abord, le domaine des *squelettes algorithmiques* sera décrit et quelques unes des nombreuses bibliothèques de ce domaine seront étudiées. Puis, quelques autres solutions à patrons seront décrites.

2.4.4.1 Squelettes algorithmiques.

Les squelettes algorithmiques parallèles ont été introduits en 1988 par Muray Cole [40]. Ils représentent des patrons de parallélisation fonctionnels, en d'autres termes des abstractions de schémas de parallélisme, que l'on retrouve de façon récurrente dans les applications parallèles. Ainsi, en théorie, n'importe quel programme parallèle peut s'exprimer comme une suite ou une imbrication de squelettes algorithmiques fonctionnels. Aucune norme n'a été définie pour écrire des squelettes, ni même aucun consensus. Toutefois, le travail de Cole [39] indique quelques règles de conception pour produire des squelettes adaptés et donc plus utilisés. Un squelette a idéalement un champ d'application le plus large possible, afin de pouvoir être utilisé dans un grand nombre de cas, sa sémantique doit être compréhensible des utilisateurs, et enfin il ne doit pas être redondant avec d'autres squelettes.

Les squelettes algorithmiques se découpent en trois grandes classes, les squelettes pour le parallélisme de données (map, reduce, zip etc.), les squelettes pour le parallélisme de tâches (farm, pipeline etc.), et enfin les squelettes dits de résolution (divide and conquer, branch and bound). Des détails sur l'ensemble de ces squelettes peuvent être trouvés dans la thèse de Legaux [84]. Nous n'allons ici décrire que quelques squelettes pour le parallélisme de données, auxquels nous feront référence dans cette thèse. Nous pouvons, tout d'abord, noter les trois squelettes de base les plus connus et les plus simples à comprendre. Le premier s'appelle *map* et permet d'appliquer une fonction *locale* à un ensemble de données d'entrée en parallèle. Une fonction locale est alors une fonction dont le calcul ne dépend que d'un élément d'entrée sans aucune dépendance avec les autres éléments. Le squelette peut alors distribuer la structure de données et appliquer la fonction à chacun des éléments séparément. Le squelette prend alors un vecteur de données d'entrée $[x_1, x_2, \ldots, x_n]$, retourne un vecteur de données de sortie $[y_1, y_2, \ldots, y_n]$ et applique une fonction f telle que

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f(x_1), f(x_2), \ldots, f(x_n)] = [y_1, y_2, \ldots, y_n].$$

Le second squelette de base est le squelette *zip* qui est une extension de *map* pour deux vecteurs d'entrée. Il distribue deux vecteurs de données d'entrée $[x_1, x_2, \ldots, x_n]$ et $[x'_1, x'_2, \ldots, x'_n]$, de même taille, et retourne un nouvel vecteur de sortie $[y_1, y_2, \ldots, y_n]$ en appliquant une fonction f telle que

$$\begin{aligned} zip\ f\ ([x_1, x_2, \ldots, x_n], [x'_1, x'_2, \ldots, x'_n]) &= [f(x_1, x'_1), f(x_2, x'_2), \ldots, f(x_n, x'_n)] \\ &= [y_1, y_2, \ldots, y_n]. \end{aligned}$$

Enfin, le squelette *reduce* permet de réduire un vecteur de données d'entrée $[x_1, x_2, \ldots, x_n]$ en un unique élément e suite à l'appel d'une opération de réduction, que nous noterons

$\oplus$, telle que

$$reduce \ \oplus \ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \ldots \oplus x_n = e.$$

Les squelettes *map* et *zip* sont des squelettes qui ne peuvent appliquer que des calculs locaux, de par leur construction. En d'autres termes, il n'est pas possible avec uniquement ces squelettes d'effectuer des calculs de type stencil. La fonction décrite par l'utilisateur ne décrit, en effet, que l'opération à effectuer sur un élément de l'ensemble de départ. Un calcul stencil dépendant d'un certain voisinage de l'élément courant, il est nécessaire de faire appel au squelette *shift*. Ce squelette va prendre un vecteur d'entrée $[x_1, x_2, \ldots, x_n]$, et retourner un vecteur de sortie $[y_1, y_2, \ldots, y_n]$ égal à l'ensemble d'entrée décalé (le décalage appliqué étant précisé par l'utilisateur). Par exemple, pour un décalage de un élément vers la droite nous obtiendront

$$[y_1, y_2, \ldots, y_n] = [\times, x_1, x_2, \ldots, x_{n-1}].$$

De cette manière en accédant au deuxième élément des ensembles $[x_1, x_2, \ldots, x_n]$ et $[\times, x_1, x_2, \ldots, x_{n-1}]$, il est possible de faire des opérations sur x_2 et x_1 en même temps.

Avec ces quatre squelettes de base, on peut très facilement observer les limites de l'approche par squelettes pour des simulations scientifiques complexes. Pour cette raison, des squelettes de type stencil sont apparus, notamment dans la bibliothèque SkelCL [21,111]. Cette bibliothèque implémente des squelettes de base pour les GPU et multi-GPU en utilisant le langage OpenCL [112]. Il s'agit donc également d'un code portable. Elle propose un squelette de stencil simple nommé *MapOverlap* qui permet de décrire une opération de stencil simple, et un squelette de stencil plus complexe, nommé *Stencil* permettant notamment de décrire des opérations stencil itératives. Afin de pouvoir effectuer les calculs de type stencil, une distribution contenant des éléments fantômes est mise en place dans la bibliothèque de squelettes, et n'existe pas dans les autres solutions. De plus, les échanges à effectuer entre les processeurs sont automatiquement détectés par les arguments utilisés dans le stencil. Notons que SkelCL ne fonctionne que pour les structures de données de type vecteur ou matrices.

Parmi les bibliothèques de squelettes permettant de faire du parallélisme de données et écrites en C++, on peut noter OSL [72], SkeTo [76], SkePu [52], et Muesli [37], chacune ayant ses propres particularités. SkeTo, par exemple, est la seule bibliothèque proposant une solution de squelettes sur les arbres [90]. SkePu propose une implémentation GPU, et Muesli une implémentation hybride MPI/OpenMP des squelettes algorithmiques de base. Enfin, OSL propose des optimisations à base de méta-programmation C++ [71,84].

Bien que les squelettes algorithmiques parallèles proposent un niveau d'abstraction intéressant, ce domaine est très peu utilisé pour des simulations scientifiques complexes. A notre connaissance, aucune simulation complexe n'a été écrite avec des squelettes algorithmiques, et leur utilisation se limite à des cas "jouet" comme la résolution de l'équation de la chaleur. Avec l'arrivée de nouveaux squelettes spécifiquement écrits pour le calcul stencil [21], l'utilisation des squelettes algorithmiques est enclin à se développer dans cette discipline. Toutefois, dans le domaine des mathématiques appliquées, les langages de programmation enseignés aux scientifiques sont très souvent des langages impératifs comme Fortran, C et C++. L'utilisation de langages fonctionnels, et donc de squelettes

algorithmiques parallèles, demande un effort d'apprentissage supplémentaire qui pourrait éloigner certains numériciens. Toutefois, notons qu'un langage fonctionnel peut être appris très rapidement et très facilement par les mathématiciens car il s'agit d'un langage plus proche des mathématiques.

2.4.4.2 D'autres solutions à patrons.

L'entreprise Google est à l'origine de la démocratisation de l'utilisation du modèle *MapReduce* [47]. Dans ce modèle il est considéré que tout calcul peut être décomposé en une série d'association du squelette *map* et du squelette *reduce*. Cette solution a permis de démocratiser les squelettes algorithmiques, grâce à l'association intelligente de deux concepts simples, dont la mise en œuvre est facilitée par un certain nombre d'outils. Ainsi, par exemple, le framework Hadoop [127], très connnu et très utilisé, propose un système de fichiers distribués et une implémentation de MapReduce. Il est alors possible de faciliter la création d'applications distribuées ainsi que leur déploiement sur des milliers de processeurs. Enfin, ce type de systèmes embarque généralement la gestion des pannes du programme ou du matériel, ce qui augmente encore l'intérêt des scientifiques, dont les simulations peuvent être très longues et coûteuses.

Google est également à l'origine d'un modèle de parallélisme simple pour les traitements parallèles sur les graphes dirigés, *Pregel* [88]. Dans l'état de l'art de ce travail, Pregel est comparé à PBGL et CGMGraph (décrits précédemment), et le principal argument avancé pour préférer son utilisation est la tolérance aux pannes. Toutefois, le type d'approche est très différent. En effet, Pregel conserve une idée proche des squelettes algorithmiques et se base également sur le modèle BSP pour structurer de façon générale des opérations sur des graphes dirigés. Dans Pregel, le graphe est tout d'abord distribué sur les différents processeurs. Un calcul dans Pregel est ensuite composé de plusieurs itérations, que l'on peut comparer à des super-étapes du modèle BSP. Dans chacune de ces étapes, le framework Pregel appelle une fonction utilisateur qu'il applique sur chaque nœud du graphe distribué. La fonction spécifie le comportement d'un unique nœud général v pour une étape S. Cette fonction peut recevoir des messages envoyés à v à l'étape $S-1$, et peut envoyer des messages à d'autres nœuds qui seront reçus à l'étape $S+1$. On retrouve alors la notion de fonction utilisateur de MapReduce, ou de tout autre squelette, et l'on retrouve également l'application de cette fonction sur chacun des nœuds, tout comme dans les squelettes algorithmiques. Toutefois le modèle Pregel est une solution plus générale que les squelettes algorithmiques habituels et ne représente pas un patron de parallélisation unique. La fonction utilisateur peut, en effet, décrire des problèmes très variés et offre une plus grande liberté de codage que dans l'utilisation des squelettes algorithmiques. Il est important de noter que les algorithmes sur les graphes peuvent être exprimés comme des chaînes d'appels à MapReduce [38, 75]. Toutefois, le modèle Pregel propose de meilleures performances. En effet, il conserve la même distribution de graphe d'une étape à l'autre du calcul, et utilise uniquement l'envoi et la réception de messages pour obtenir les informations d'autres processeurs. L'utilisation de MapReduce implique, quant à elle, tout d'abord (1) une distribution initiale des données, puis (2) l'application du map, puis pour terminer (3) des communications pour

l'application du reduce, ce qui revient à communiquer toutes les données résultantes du map à chaque appel d'un mapreduce. Giraph [7] est une alternative à Pregel et utilise les même concepts.

Là encore il s'agit de solutions de parallélisme implicite très intéressantes et avec un certain nombre d'avantages. L'utilisation de ce type de solutions semble, tout d'abord, simplifier encore davantage la création de programmes parallèles, et la classification des différentes opérations d'un programme en patrons ne semble pas extrêmement difficile à élaborer. De plus, ces solutions utilisent deux types de spécificités pour mettre au point des optimisations : les structure de données distribuées, et les patrons utilisés. Les possibilités de performances paraissent donc plus importantes que dans les solutions dites générales. Cependant, un certain nombre de problèmes peuvent être notés avec ce type de solution. Tout d'abord, et comme nous l'avons décrit, chaque opération décrite à l'aide d'un patron, ou d'un squelette, est en fait une opération simple. À l'exception de Pregel, ces solutions sont très proches de la programmation fonctionnelle. Ainsi, si des calculs complexes doivent être mis en œuvre, un grand nombre d'appels imbriqués sera nécessaire, ce qui peut complexifier l'écriture, la lecture et la compréhension des programmes. Dans cette solution, de nouveau, il semble possible que la difficulté de programmation parallèle soit déportée vers l'utilisation de nouveaux concepts, et notamment vers les difficultés de la programmation fonctionnelle, non connue de la plupart des scientifiques.

2.4.5 Solutions spécifiques à un domaine

Nous avons donc vu que les solutions générales de parallélisme implicite sont uniquement spécifiques aux types de conteneurs utilisés, ce qui peut limiter les optimisations du programme parallèle. Les solutions à patrons de parallélisme sont, quant à elles, spécifiques aux types de conteneurs mais aussi aux types de patrons utilisés, ce qui augmente les possibilités d'optimisations pour le problème posé. Dans cette dernière partie, nous allons voir le niveau d'abstraction et de spécificité le plus haut qui est proposé dans les solutions de parallélisme implicite pour le calcul scientifique. Dans ce cas, la solution, qu'elle soit une bibliothèque, un framework ou un *langage dédié* (noté DSL), est spécifiquement implémentée pour un problème spécifique et propose des optimisations de performances dues à cette spécificité, qu'on ne pourrait donc pas retrouver dans les solutions plus générales. La définition de "spécifique" peut être très variée. Par exemple, STAPL peut être vu comme un langage dédié à la programmation par conteneurs parallèles. Toutefois, nous rattacherons ici, et dans le reste de cette thèse, une solution dite spécifique (de type DSL, bibliothèque etc.) à une solution spécifique au calcul scientifique. Le domaine du calcul scientifique reste un domaine très vaste, même si il est beaucoup plus spécifique que les domaines traités par STAPL, PBGL ou les squelettes algorithmiques. Pour cette raison, de nombreuses solutions spécifiques ont été mises au point pour divers sous-problèmes du calcul scientifique. Certaines solutions sont plus spécifiques que d'autres, tout le problème étant de trouver le niveau d'abstraction idéal pour l'utilisation qui sera faite du langage.

Parmi les solutions parallèles spécifiques aux applications scientifiques, on peut tout

d'abord noter ScaLAPACK [19] qui est une bibliothèque haute performance pour les calculs de l'algèbre linéaire, implémentée pour les architectures parallèles à mémoire distribuée. On peut ensuite noter FFTW [55], pour le calcul parallèle de la transformée de Fourier discrète, et donc notamment pour des problèmes de traitement du signal. Ce DSL est implémenté pour des architectures à mémoire partagée et distribuée. SPIRAL [104] est, quant à lui, un DSL plus récent permettant, de façon plus générale que FFTW, d'effectuer des traitements du signal numérique.

Dans cette thèse, nous nous intéressons plus particulièrement à la résolution des EDP par des méthodes numériques basées sur des maillages, et donnant lieu à des schémas numériques explicites. Nous nous intéressons donc aux problèmes stencils pour tout type de maillages, et nous allons évoquer avec plus d'attention, dans le reste de cette section, les DSL, bibliothèques et frameworks spécifiquement développés pour ce type de calculs que l'on peut appeler plus généralement des *DSS* pour *Domain Specific Solutions*.

2.4.5.1 EDP et EDO sur maillages structurés

Commençons par évoquer les solutions de parallélisme implicite spécifiques au calcul de type stencil sur les maillages structurés. Il en existe en très grand nombre, et il ne s'agit pas ici d'une liste exhaustive de ces solutions mais des quelques travaux qui paraissent proches des solutions présentées dans cette thèse. Notons tout d'abord l'une des solutions les plus utilisées et les plus connues du monde de la simulation scientifiques, PETSc [10–12]. PETSc est une solution très vaste qui permet d'écrire des applications scientifiques, modélisées par des EDP, en parallèle. Cette solution est composée d'outils permettant d'effectuer des opérations sur des vecteurs et des matrices, de solveurs d'équations linéaires et non linéaires, mais également d'outils graphiques permettant la visualisation des résultats de l'application. PETSc est implémenté pour les architectures à mémoire partagée CPU et GPU grâce aux modèles *pthreads*, CUDA et OpenCL, pour les architectures à mémoire distribuée, grâce à l'utilisation de MPI, et pour les architectures à mémoire hybrides aux travers des associations MPI-pthreads et MPI-GPU. PETSc est basé sur un ensemble de structures de données distribuées et sur un ensemble de fonctions ou routines spécialisées pour ce type de traitements. PETSc est donc identifiable à une bibliothèque sur conteneurs, tout comme STAPL ou PSTL, mais dont les interfaces et les algorithmes sont spécifiques au traitement des EDP. Nous pouvons également noter la bibliothèque spécifique Trilinos [68] qui est très proche de PETSc.

Nous pouvons également noter des solutions spécifiques plus récentes, comme par exemple le framework développé à Berkeley permettant de générer automatiquement un code parallèle, adapté à l'architecture à mémoire partagée et au matériel utilisé (dit *auto-tuned*), uniquement à l'aide de l'expression d'un stencil en Fortran [74]. Dans cette même famille de framework auto-tuned, on peut également noter PATUS [36] qui génère du code parallèle CPU et GPU à partir de l'expression d'un stencil et d'une stratégie de parallélisation. PATUS évalue ensuite la meilleure parallélisation pour le matériel utilisé. Évoquons également Panorama [86] qui utilise des techniques particulières pour minimiser les défauts de cache, et Pochoir [116] qui permet la définition de stencils à n dimensions en C++. Physis [89] ne propose pas de solution auto-tuned mais permet lui aussi de

définir une expression stencil à l'aide d'un DSL, et d'en produire automatiquement des applications parallèles MPI et CUDA.

2.4.5.2 EDP sur maillages non-structurés

De nombreuses solutions de parallélisme implicte, spécifiques aux calculs de type stencil, sont donc disponibles pour les maillages structurés. Lorsque le problème des maillages non-structurés est abordé, les outils se font plus rares, mais il en existe également. Nous pouvons, tout d'abord, évoquer le plus ancien d'entre eux, OP2 [59,60,94]. OP2, développé à l'université d'Oxford, est une révision du framework OPlus [23], initié en 1993, qui permettait d'écrire des applications basées sur un maillage non-structuré et sur la méthode des éléments finis. OPlus a notamment été utilisé pour paralléliser, en 1995, une simulation d'écoulement des fluides non visqueux sur un maillage complexe représentant un avion [45]. OPlus était implémenté pour les architectures à mémoire distribuée et implémenté en MPI. OP2 est une version plus moderne et plus récente de OPlus qui est implémentée pour les architectures à mémoire partagée CPU et GPU. Le framework OP2 charge l'utilisateur de quatre parties : (1) définir des ensembles d'éléments qui vont définir les éléments du maillage, (2) définir des liens entre ces ensembles pour former le maillage, (3) définir des données sur les ensembles, et enfin (4) implémenter des opérations sur les ensembles d'éléments. Il est ainsi possible de définir un maillage non-structuré de toute forme, ainsi que sa topologie. Le framework dispose d'un compilateur permettant de transformer le code OP2 en un code C++, qui pourra à son tour être compilé. Le DSL Liszt [50] permet également de coder des simulations sur maillages non-structurés en parallèle. Toutefois, le niveau d'abstraction proposé à l'utilisateur est légèrement différent. En effet, le niveau d'abstraction de OP2 est plus proche du niveau d'abstraction des squelettes algorithmiques puisque l'utilisateur n'a pas à définir ses boucles, alors que Liszt permet de rester plus proche d'un code séquentiel avec la gestion des boucles par l'utilisateur. Le langage Liszt est une sous-partie du langage de programmation Scala [99], et utilise une version modifiée de son compilateur. Liszt supporte une implémentation MPI, OpenMP et CUDA/OpenCL.

Dans une solution de parallélisme implicite spécifique, comme celles qui ont été évoquées ici, il est nécessaire de doser convenablement le niveau de spécificité de la solution. D'une part, si le niveau de spécificité est trop important, cela peut nuire au nombre d'utilisateurs. D'autre part, et à l'inverse, si le niveau de spécificité est trop faible, la solution peut s'avérer trop généraliste et risque de ne pas répondre aux attentes des utilisateurs. Toutefois, en trouvant un niveau de spécificité adéquat, ces solutions sont souvent celles qui procurent les meilleures performances, et la plus grande notoriété auprès des utilisateurs non-informaticiens.

2.5 Calculs de performances et difficulté de programmation

Dans cette thèse est proposée l'implémentation d'une solution de parallélisme implicite pour les simulations basées sur des maillages. L'évaluation de cette implémentation passe donc par deux volets, tout d'abord l'évaluation des performances produites par la solution de parallélisme implicite, mais également l'évaluation de la difficulté de codage liée à l'utilisation cette solution. Il existe un lien fort entre les performances d'une solution de parallélisme implicite et sa simplicité de codage, puisque le fait de cacher des opérations parallèles peut engendrer un certain sur-coût. Une solution de parallélisme implicite idéale sera à la fois performante et très simple d'utilisation. Cette dernière section de notre état de l'art va donc tout d'abord introduire les mesures de performances, puis les métriques d'effort utilisées dans cette thèse.

2.5.1 Mesures de performances

En science informatique, on appelle *benchmarking* une méthode permettant de quantifier et d'évaluer les résultats expérimentaux d'un code ou d'un programme. Plus particulièrement, en calcul parallèle, le benchmarking est souvent associé aux méthodes permettant d'évaluer les performances d'un programme, de façon absolue, ou de façon relative à un autre programme. L'idée de base est de calculer le temps d'exécution d'un programme, ou d'une sous-partie du programme, représentative du problème à évaluer. Ce temps d'exécution peut ensuite être utilisé pour évaluer plusieurs types de métriques comme le temps d'exécution total et moyen du programme, l'accélération du programme, le débit des échanges de données par le programme (exprimés en bits par seconde), ou encore la puissance du programme (exprimé en nombre d'opérations flottantes, par seconde). Certaines de ces métriques peuvent être comparées de façon absolue avec un idéal de référence. C'est le cas, par exemple, de la puissance du programme qui ne peut en théorie pas dépasser les capacités matérielle des machines utilisées. D'autres métriques, en revanche, ne sont utiles que dans le cas d'une comparaison avec d'autres programmes, comme par exemple le temps d'exécution.

Dans cette thèse deux métriques en particulier seront utilisées pour évaluer les performances des solutions proposées. La première est la représentation de l'accélération d'un programme, qui nous permettra d'évaluer la montée en charge des programmes parallèles implémentés. Nous appellerons cette métrique la *scalabilité*. Il existe deux méthodes pour évaluer la scalabilité d'un programme parallèle. La première est appelée la scalabilité faible, la seconde la scalabilité forte. Notons $T(p, n)$ le temps nécessaire pour exécuter un programme en utilisant p processeurs et pour une taille de problème n. On définit alors l'accélération du programme, pour un problème de taille n et pour p processeurs, par

$$speedup(p, n) = \frac{T_{seq}(1, n)}{T(p, n)}, \tag{2.19}$$

où $T_{seq}(1,n)$ représente le meilleur temps séquentiel connu pour la résolution du problème courrant, alors que $T(p,n)$ représente le temps du programme parallèle à évaluer. Il ne s'agit donc pas du même programme et le temps $T_{seq}(1,n)$ est considéré comme la référence du problème. Cette définition de l'accélération permet d'évaluer une scalabilité dite forte et comparable entre différentes verions parallèles. La taille du problème n'est pas modifiée entre l'exécution séquentielle (sur un unique processeur) et l'exécution parallèle (sur p processeurs). Toutefois, étant donné qu'il est nécessaire de disposer de $T_{seq}(1,n)$ pour évaluer cette l'accélération, une version modifiée de cette définition est généralement utilisée, et sera utilisée dans cette thèse. L'accélération est alors définie par

$$speedup(p,n) = \frac{T(1,n)}{T(p,n)}. \tag{2.20}$$

Dans ce cas c'est le temps séquentiel du programme à évaluer qui est utilisé comme temps de référence. Cette accélération est moins intéressante et ne permet pas une comparaison intéressante de deux versions parallèles différentes. Toutefois, elle permet d'observer la scalabilité du programme à évaluer.

Une deuxième définition de l'accélération du programme, pour un problème de taille n et pour p processeurs, est

$$speedup(p,n) = \frac{T(1,n)}{T(p,p \times n)}.$$

Dans ce cas, la scalabilité évaluée est dite faible car la taille du problème est multipliée par le nombre de processeurs utilisés. Ainsi, la quantité de travail assignée à chaque processeur reste constante et le nombre de processeurs utilisés (et donc la taille du problème général) augmente. Cette accélération est idéale si le temps de calcul reste constant avec l'augmentation du nombre de processeurs et de la taille du problème. Cette accélération peut être utile dans plusieurs cas. Tout d'abord si le programme parallèle contient une fraction de code séquentielle et une fraction de code parallèle, cette accélération peut donner des indications sur le temps représenté par la fraction séquentielle et sur la nécessité de réduire cette fraction. Une fraction de code parallèle que nous appelons *classique*, c'est à dire utilisant relativement peu de communications (les plus proches voisins par exemple), ne rencontre pas de problème de scalabilité dans le cas d'une accélération faible. Si, en revanche, le programme parallèle utilise des communications très lourdes, telles que des communications collectives dont le coût augmente avec le nombre de processeurs, certains problèmes peuvent être détectés en représentant cette accélération. Enfin, si le programme consomme beaucoup de mémoire et ne peut, par exemple, pas être exécuté en séquentiel, cette accélération peut permettre d'évaluer tout de même son accélération. Dans cette thèse, nous utiliserons la définition de l'accélération forte modifiée (2.20) pour évaluer la scalabilité des programmes.

Si l'on évalue l'accélération (2.20) d'un programme parallèle en utilisant P processeurs, on représente généralement une courbe de l'ensemble des valeurs de l'accélération entre 1 et P processeurs. Cette courbe représente donc l'accélération du programme parallèle en fonction du nombre de processeurs utilisés. Étant donné la définition de l'accélération (2.20), il peut être déduit que l'accélération idéale d'un programme est égale

à p pour tout p dans $[1, P]$. L'accélération idéale est alors représentée par la fonction $f(p) = p$. Il en résulte qu'un bon speedup sera idéalement le plus proche possible de la droite $f(p) = p$, et idéalement linéaire quelque soit le nombre de processeurs utilisés. Toutefois la réalité sur l'accélération d'un programme peut être différente de ces déductions théoriques. Il est en effet possible de dépasser l'accélération idéale théorique. C'est ce qu'on appelle une accélération super-linéaire. Ce phénomène peut être dû à plusieurs facteurs. La première raison concerne les architectures à mémoire distribuées. Imaginons alors que le programme séquentiel utilise trop de mémoire, il est alors possible que les données du programme ne tiennent plus en mémoire vive et soient stockées sur la mémoire disque de la machine (ce qu'on appelle du *swapping*). Dans ce cas, le temps d'exécution séquentiel peut être anormalement long. Ainsi, en augmentant le nombre de processeurs, et donc en réduisant l'emprunte mémoire du programme pour chaque processeur, l'accélération peut être facilement supérieure à p. La deuxième raison qui peut être à l'origine d'une super-linéarité est proche de la première mais non restreinte aux architectures à mémoire distribuée. Elle ne concerne plus le passage de la mémoire vive à l'espace disque, mais le passage de la mémoire cache à la mémoire vive. En réduisant la taille du problème (du fait du nombre de processeurs utilisés), il peut arriver que la taille des structures de données sur lesquelles sont effectués les calculs soit inférieure ou égale à la taille des lignes de cache, ce qui réduit le nombre d'accès à la mémoire vive depuis le cache, et ce qui augmente l'efficacité du programme. Cependant, lorsqu'un phénomène de cache se produit et qu'il conduit à une accélération super-linéaire, il peut être intéressant de modifier l'algorithme séquentiel afin que celui-ci traite des blocs de données plus petits, limitant ainsi les chargements de données en cache pendant les calculs. De cette façon, la super-linéarité est souvent réduite et l'accélération observée sera probablement plus réaliste.

L'évaluation de la scalabilité forte d'un programme est un bon indicateur des performances du programme parallèle. Toutefois, l'accélération définie par (2.20), et utilisée dans cette thèse, souffre de certaines faiblesses lorsque l'on cherche à comparer les performances de deux implémentations différentes. Tout d'abord l'accélération d'un programme est liée au temps d'exécution séquentiel du programme. On ne peut donc pas comparer l'accélération de deux implémentations n'ayant pas le même temps d'exécution séquentiel. De même, meilleure est l'implémentation, plus difficile est l'obtention d'une bonne accélération, car la version moins optimisée passera plus de temps dans les calculs et aura probablement une accélération linéaire sur un plus grand nombre de processeurs. Une accélération est donc un bon indicateur de scalabilité pour un programme, mais la définition (2.20) n'est pas une bonne métrique pour comparer deux implémentations différentes. Pour cette raison nous utilisons une deuxième métrique dans cette thèse. Elle représente simplement le temps d'exécution des programmes et nous permettra de comparer de façon objective les temps d'exécution de plusieurs implémentations parallèles d'une même simulation scientifique. Notons que nous avons fait le choix, dans la plupart de nos résultats, de représenter les temps d'exécution avec une échelle logarithmique. Cette échelle nous permet, tout d'abord, de faciliter la lisibilité des temps d'exécution,

mais permet également de comparer à la fois les temps d'exécution et la linéarité des accélérations des implémentations.

2.5.2 Effort de programmation

Il existe diverses métriques permettant d'évaluer l'effort à fournir pour écrire un code. Certaines métriques permettent d'évaluer la difficulté d'un programme à partir du nombre de fonctionnalités à implémenter [4,80]. Ce type de métriques est utilisé dans la conception et le génie logiciel, mais ne nous sera pas utile pour comparer deux versions parallèles possédant les mêmes fonctionnalités. La *complexité cyclomatique* [91] est une métrique qui base la difficulté de programmation sur le comportement d'un programme. Elle est basée sur un graphe qui représente les différentes exécutions possibles d'un même programme, pour en estimer sa complexité. En d'autres termes, cette métrique s'intéresse aux branches conditionnelles du programme. De nouveau, nous ne pourrons utiliser cette métrique pour comparer deux implémentations d'une même simulation. Nous pouvons noter deux métriques qui pourraient être utilisées dans notre cas. Tout d'abord, la métrique SLOC (*Source Lines of Code*) se base uniquement sur le nombre de lignes dans un code pour évaluer la difficulté d'un programme. Il s'agit d'une première indication sur l'effort à fournir pour écrire un programme. Mais nous allons plus particulièrement nous concentrer sur les métriques de Halstead [65], qui offrent des indicateurs plus révélateurs sur l'effort de programmation à fournir pour écrire un code.

Les métriques de Halstead sont basées sur le dénombrement des opérateurs et des opérandes d'un code source. De ce dénombrement, directement appliqué dans le code, sont obtenues quatre mesures représentées dans la table 2.2. Afin d'évaluer correctement ces mesures il est important de définir ce que l'on considère comme un opérateur et un opérande. Peu d'informations sur ce sujet sont données dans la littérature. Dans nos codes C++, nous avons considéré que les opérandes étaient l'ensemble des variables et constantes définies par l'utilisateur. Les opérateurs sont l'ensemble des opérations numériques, affectations et opérateurs de comparaison (+,∗,−,/,=,==,&&,<= etc.), l'ensemble des mots clés du C++ (*static*, *class*, *template* etc.), l'ensemble des types (*int*, *const*, $float$, ∗ etc.), l'ensemble des instructions du C++ (for, *while*, *do*, $if/elseif/else$ etc.), les symboles délimiteur ;, les parenthèses et les appels de fonctions.

Symbole	Mesure
N_1	Nombre total d'opérateurs
N_2	Nombre total d'opérandes
η_1	Nombre d'opérateurs distincts
η_2	Nombre d'opérandes distincts

TABLE 2.2 – *Mesures directes dans le code*

Grâce à ces quatre mesures, les métriques de Halstead peuvent être calculées et sont représentées dans la table 2.3. La première représente le vocabulaire du programme, la deuxième la longueur du programme, qui n'est pas directement liée aux nombre de lignes

de code mais au nombre total d'opérandes et d'opérateurs. La troisième métrique représente le volume du programme. Ce volume est le produit de la longueur du programme et du logarithme en base deux du vocabulaire. Comme ce volume est basé sur le nombre d'opérations effectuées et d'opérandes gérées dans le programme, il est moins sensible à la disposition du code que les mesures SLOC. La métrique suivante représente la difficulté, et la propension à l'erreur, d'un programme. Cette métrique est calculée comme un produit entre le vocabulaire des opérateurs et la fréquence d'apparition des opérandes. Elle part donc du principe que plus le nombre d'opérateurs distincts est grand, plus il est difficile d'implémenter le programme, et que plus les même opérandes sont utilisées dans le programme, plus la propension à l'erreur est grande. Enfin la dernière métrique représente l'effort nécessaire à l'écriture du programme et est égale au produit du volume par la difficulté. Ainsi, plus un programme est volumineux et difficile, plus l'effort de programmation à fournir sera important.

Symbole	Valeur	Métrique
η	$\eta_1 + \eta_2$	Vocabulaire
N	$N_1 + N_2$	Longueur
V	$N \times \log_2 \eta$	Volume
D	$\frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$	Difficulté
E	$D \times V$	Effort

TABLE 2.3 – *Métriques de Halstead*

Les métriques de Halstead proposent donc des concepts intéressants et permettent de pouvoir comparer deux implémentations différentes, en terme d'effort de programmation, ce qui n'est pas le cas des autres métriques. Même si ces métriques ont été proposées pour des programmes séquentiels, elles s'appliquent, de notre point de vue, à des programmes parallèles. Toutefois, notons que ces métriques comportent des faiblesses pour exprimer l'effort de programmation d'un programme parallèle. Tout d'abord, dans un programme parallèle, un effort plus important est demandé aux utilisateurs pour utiliser des opérateurs parallèles et des opérandes distribuées, que pour utiliser des opérateurs et opérandes classiques de la programmation séquentielle. De plus, dans la conception d'un programme parallèle, les appels à des opérateurs parallèles (comme par exemple les routine de MPI), et l'utilisation d'opérandes distribuées ne sont pas les seuls difficultés. En effet, l'un des points les plus difficiles dans la programmation parallèle est la conception du programme. C'est, en effet, le développeur qui doit réfléchir à la façon dont le programme va pouvoir fonctionner en parallèle et ce n'est pas une difficulté qui peut transparaître dans le dénombrement des opérandes et des opérateurs. Il paraît très difficile de pouvoir évaluer ce type de difficulté, aussi les métriques de Halstead restent, de notre point de vue, les métriques les plus adaptées à l'utilisation que nous souhaitons en faire dans cette thèse.

2.6 Conclusion et positionnement du travail

Cette thèse vise à proposer des solutions de parallélisme implicite pour le cas spécifique des simulations numériques scientifiques. Pour cette raison, cet état de l'art a, tout d'abord, évoqué les architectures parallèles, les paradigmes et les modèles de programmation et leur évolution au fil du temps, ainsi que la discrétisation et les méthodes numériques de résolution des EDP. Dans les concepts introduits dans la section 2.2 de cet état de l'art, nous nous intéressons plus particulièrement aux résolutions d'EDP basées sur des maillages quelconques, en utilisant les méthodes numériques introduites dans la partie 2.2.4. Le travail présenté dans cette thèse se limite, dans les chapitre 3 et 4, aux calculs de schémas numériques explicites (2.3), toutefois le chapitre 5 évoquera et traitera le cas de schémas numériques implicites (2.4) également. Cette thèse propose des modèles et des implémentations basés sur le paradigme de parallélisme de données et sur le paradigme SPMD. Pour cette raison, le problème de partitionnement des données est un point à aborder et que nous traitons plus particulièrement dans le chapitre 5. Nous présentons dans cette thèse un modèle de programmation implicite nommé SIPSim (pour *Structured Implicit Parallelism for Scientific SIMulations*), puis par la suite son implémentation pour des architectures à mémoire distribuée, nommée *SkelGIS*, qui permet de valider le modèle. L'approche SIPSim s'applique, a priori, à tout type de maillage, toutefois cette thèse s'intéresse à l'implémentation de deux cas particuliers : les maillages cartésiens à deux dimensions et la composition de maillages sous forme de réseaux. Le modèle SIPSim permet de générer des programmes parallèles SPMD du type de l'algorithme 2, en conservant une programmation séquentielle comme introduite dans l'algorithme 1. Enfin, l'implémentation actuelle de l'approche SIPSim (SkelGIS) est effectuée en utilisant le modèle MPI, introduit dans cet état de l'art.

C'est dans la partie 2.4 de cet état de l'art qu'a été introduit le cœur de cette thèse en décrivant les modèles et les solutions de parallélisme implicite. Les avantages et les inconvénients de chaque vision du parallélisme implicite ont été donnés et analysés, ce qui nous permet de positionner notre travail dans ce contexte. La figure 2.12 résume ce positionnement, que l'on peut aussi résumer ainsi :

- Des bibliothèques générales de parallélisme implicite, notre travail tente de conserver une certaine flexibilité, ou souplesse, qui permet de répondre notamment à des cas particuliers de simulations. Nous héritons par exemple du concept très important d'*itérateur* de STAPL ou de PSTL, sous une forme différente.
- Des solutions à patrons, et des bibliothèques de squelettes algorithmiques, notre travail hérite d'un haut niveau d'abstraction. L'utilisateur définit, en effet, des fonctions séquentielles qui sont appliquées au travers de patrons où les communications entre les processeurs lui sont cachées. Nos travaux sont notamment proches des concepts introduits par le modèle Pregel, proche de BSP.
- Enfin, des langages et bibliothèques spécifiques, notre travail cherche à retrouver une efficacité propre aux problèmes spécifiquement traités, par le biais d'optimisations. Le framework OP2 et le DSL Liszt sont les solutions spécifiques les plus proches de nos travaux.

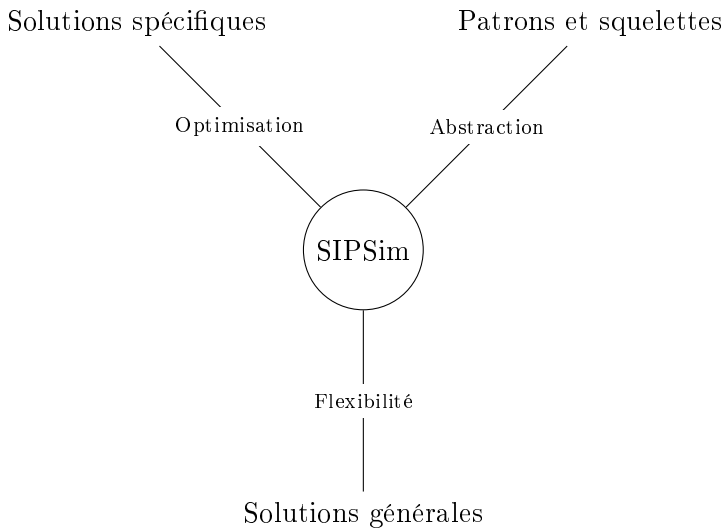


FIGURE 2.12 – *Placement de notre travail par rapport à l'existant.*

Enfin, afin de comprendre les résultats présentés dans cette thèse, nous avons terminé cet état de l'art par une présentation des mesures de performance et des mesures de difficulté de programmation.

SIPSim : Structured Implicit Parallelism for scientific Simulations

3

Sommaire

Dans cette thèse nous nous intéressons aux simulations scientifiques dont les équations aux dérivées partielles sont résolues par des méthodes numériques qui discrétisent l'espace et le temps. On appelle ces simulations des simulations basées sur des maillages. Nous nous intéressons plus précisément, et dans un premier temps, aux simulations dont les schémas numériques sont explicites et donc de la forme de l'équation (2.3) présentée dans la section 2.2.3.2. En informatique, ce type de calcul est appelé un calcul *stencil*. Le parallélisme implicite pour des calculs de type *stencil* est un domaine très actif de la recherche en informatique. Dans ce chapitre est présentée la méthode *SIPSim*, qui signifie *Structured Implicit Parallelism for scientific Simulations*. SIPSim permet d'obtenir une vision systématique des besoins pour proposer une solution de parallélisme implicite pour ce type de simulations. SIPSim peut donc être considéré comme un modèle de programmation parallèle implicite pour les simulations scientifiques.

Afin de définir une approche pertinente pour élaborer des solutions de parallélisme implicite pour les simulations scientifiques, il faut tout d'abord étudier la parallélisation de ces simulations scientifiques. Comme nous l'avons déjà décrit dans l'état de l'art 2.4, Pingali et Al [103] ont défini "The TAO of Parallelism in Algorithms", qui propose une classification intéressante des différents types de problèmes à paralléliser. Une fois un problème classifié dans le "TAO", des solutions connues de parallélisation peuvent être appliquées. La parallélisation est donc facilitée grâce à cette classification, mais en aucun cas cachée, comme nous cherchons à le faire. Comme nous l'avons déjà détaillé dans la section 2.4.1 de l'état de l'art, le type de simulations auxquelles nous nous intéressons dans ce travail (sur maillages fixes et schémas numériques explicites) sont classifiées dans le "TAO" comme des algorithmes *topology-driven*, dont l'ensemble des éléments du maillage sont identifiés comme les nœuds actifs (*active nodes*) de l'algorithme, et peuvent être traités de façon non-ordonnée dans l'algorithme. Enfin, les calculs sont considérés comme locaux car ne modifiant pas le maillage d'entrée.

Comme il l'a déjà été évoqué précédemment, pour des architectures parallèles à mémoire distribuée, ce genre de simulations est généralement parallélisé en utilisant l'approche SPMD (Simple Program Multiple Data) décrite dans l'état de l'art. Cette approche se prête bien aux simulations basées sur les maillages puisqu'elle consiste alors à partitionner le maillage en plusieurs parties, chacune confiées à des processeurs différents qui exécuteront le même code sur leur sous-partie du maillage. L'algorithme 2 de la partie 2.2.5 illustre ce type de parallélisation, et représente la base de l'analyse de l'approche SIPSim. Nous rappelons ici cet algorithme avec plus de détails. Nous notons S_b le schéma numérique à appliquer aux éléments de la bordure physique du maillage, qui correspond donc à calculer les conditions limites. Nous notons, de plus, S le schéma numérique permettant le calcul des quantités pour les autres éléments du maillage.

Cet algorithme parallèle peut très clairement être apparenté au modèle BSP, introduit lui aussi dans l'état de l'art. En effet, dans cet algorithme peuvent être identifiées trois super-étapes. Tout d'abord une super-étape de communication est effectuée au début de chaque itération de temps. Dans cette étape, chaque processeur reçoit les valeurs sur le voisinage $N(x)$, qu'il ne possède pas dans son sous-maillage, afin de pouvoir calculer, de façon correcte, l'ensemble des nouvelles valeurs pour la nouvelle itération de temps.

Algorithme 3 : Algorithme parallèle SPMD d'une simulation basée sur un maillage.

```
Création du maillage μ
Partitionnement du maillage μ = {μ_0, μ_1, ... μ_{p-1}}
Création des quantités à simuler appliquées à μ
Initialisation des quantités et des paramètres
Définition du pas de temps, commun à tous les processeurs : t
Définition du temps maximal, commun à tous les processeurs : tmax
tant que t<tmax faire
    Communication de N(x) entre les processeurs
    pour chaque b ∈ E_{μ_i} de la bordure physique faire
        Obtention de σ(b, t-1) et σ(y, t-1), y ∈ N(b)
        Calcul du schéma S_b(b, t)
    fin
    pour chaque x ∈ E_{μ_i} faire
        Obtention de σ(x, t-1) et σ(y, t-1), y ∈ N(x)
        Calcul du schéma S(x, t)
    fin
    t = t+1
fin
```

Il effectue également, dans cette super-étape, l'envoi des valeurs des éléments dont les autres processeurs ont besoin. La seconde super-étape consiste à calculer le schéma numérique à appliquer aux éléments de la bordure physique du domaine S_b. Enfin, la dernière super-étape consiste à calculer le schéma numérique S. Ces deux dernières super-étapes peuvent être exécutées de façon asynchrone par l'ensemble des processeurs. En effet, chacun des processeurs, ayant reçu dans la première super-étape les valeurs nécessaires à l'ensemble des calculs, ils sont en mesure d'exécuter de façon autonome ces super-étapes. En d'autres termes, il est possible d'affirmer qu'une simulation scientifique parallèle, basée sur un maillage et des schémas numériques explicites, peut être identifiée à un programme parallèle BSP contenant une super-étape de communications et n super-étapes de calculs, où n est le nombre de schémas numériques à appliquer à chaque itération de temps.

Dans une solution de parallélisme implicite, le but est d'identifier quelles parties du programme peuvent être conservées comme séquentielles, et n'ont donc pas besoin d'être cachées à l'utilisateur, et quelles parties du programme sont nécessairement parallèles et nécessitent donc de devenir implicites pour l'utilisateur. Dans le modèle BSP, chaque super-étape de calcul peut être considérée comme un programme séquentiel qui sera exécuté sur une sous-partie des données initiales. Pour cette raison, les n super-étapes de calcul, dans une simulation scientifique, sont des calculs séquentiels, que l'utilisateur est capable de renseigner. De l'algorithme 3, on peut alors déduire que quatre composants

principaux sont nécessaires à la création d'une solution de parallélisme implicite pour les simulations scientifiques :

— Un maillage distribué de façon implicite
— Une création des quantités à simuler, implicitement distribuées sur le maillage
— Un moyen implicite de cacher les communications entre les processeurs
— Des interfaces de programmation pour manipuler de façon implicite les données distribuées

Ces quatre composants forment l'approche systématique SIPSim et sont détaillés dans le reste de ce chapitre. Ces composants sont nommés, *structure de données distribuée* (DDS), *application de données* (DPMap), *applicateur* (AP) et *interfaces de programmation* (PI).

3.1 Structure de données distribuée

La structure de données distribuée, ou DDS, est le composant le plus complexe de l'approche SIPSim. Il est chargé de représenter la structure de données efficace qui stocke le maillage, et qui conserve également les informations sur sa connectivité, et tout ceci en essayant de réduire, autant que possible, son empreinte mémoire. Il est également responsable de distribuer ou partitionner le maillage sur les différents processeurs. Ce composant sera donc, en très grande partie, responsable de l'efficacité de la solution, puisque qu'il sera de façon sous-jacente responsable de

— l'accès efficace aux éléments du maillage,
— l'accès efficace au voisinage $N(x)$ des éléments du maillage,
— la répartition équilibrée de la charge de travail sur les processeurs,
— la minimisation du nombre de communications nécessaires entre les processeurs.

Le principal effort de programmation à fournir, pour proposer une solution SIPSim efficace, se trouve dans ce composant.

Essayons maintenant de formaliser sa définition. Une structure de données DS définit l'implémentation d'une structure où le maillage μ est représenté. Une structure de données devant être efficace dans l'accès aux éléments, il est souvent nécessaire d'y stocker plus d'informations que la simple connectivité du maillage. Pour cette raison, nous noterons $\mu \subset DS$ le fait que le maillage μ soit implémenté par DS. Comme expliqué précédemment, la version parallèle SPMD de la simulation découpe, ou partitionne, le maillage en p parties $\mu_0, \ldots, \mu_{p-1}$, où p est le nombre de processeurs. Dans ce cas, la structure de données doit être distribuée telle que $DDS = \cup_{i=0}^{p-1} DDS_i$, où pour tout $i \in [0, p-1], \mu_i \subset DDS_i$. On peut également noter que la DDS est, a priori, différente de DS car les structures de données DDS_i contiennent généralement plus d'informations que DS, comme par exemple des informations sur les communications, ou les mailles fantômes nécessaires au calcul.

Deux informations importantes doivent être prises en compte pour implémenter une DDS efficace avec l'approche SIPSim. Tout d'abord, afin de répondre aux deux premiers points évoqués plus haut, il sera nécessaire de réfléchir à l'implémentation de la structure

de données en elle-même pour permettre, idéalement, un accès aux éléments et à leur voisinage en $O(1)$. De plus, une attention particulière devra être portée sur la distribution du maillage sur les différents processeurs. Un maillage peut être considéré comme un graphe de plusieurs façons [46,106,122], comme cela a été introduit dans la section 2.3.3 de l'état de l'art. Soit en associant chaque intersection et chaque côté d'une maille respectivement à un nœud et à une arête d'un graphe, soit en considérant chaque cellule du maillage comme un nœud du graphe, soit en combinant plusieurs approches etc. Par conséquent, la distribution d'un maillage revient à un problème de partitionnement de graphe, qui est connu pour être NP-complet de l'état de l'art. Pour les maillages cartésiens réguliers des solutions plus simples et plus efficaces peuvent toutefois être mises en place, comme un découpage rectiligne en lignes, en colonnes ou en blocs (voir la section 2.3). Toutefois, pour des maillages plus complexes et irréguliers, le problème de partitionnement doit être étudié afin d'obtenir une solution SIPSim efficace. Un bon partitionnement sera, en effet, l'une des clés de l'efficacité de la solution car la super-étape de communication synchronise les processeurs entre eux. Il ne serait pas idéal, en termes de performances, que des processeurs attendent et que d'autres soient surchargés de travail.

3.2 Application de données

Un objet d'application de données, ou DPMap, consiste à permettre d'appliquer les quantités à simuler sur le maillage distribué de façon implicite pour l'utilisateur. Il existe plusieurs façons d'appliquer des données sur un maillage. La plus simple, est de directement embarquer les données dans une instance de l'objet DDS et d'en créer autant d'instances que de quantités à simuler. C'est une solution simple à mettre en place et qui peut être efficace si l'implémentation du DDS est légère et si la connectivité du maillage est elle même une structure de données simple, comme un tableau. En d'autres termes, il peut s'agir d'une bonne solution pour des maillages réguliers et simples. En revanche lorsque le maillage se complexifie et que sa connectivité est implémentée de façon plus complexe et lourde, par plusieurs tableaux par exemple, cette solution peut devenir très coûteuse en mémoire puisqu'à chaque création de quantité à simuler, le DDS est dupliqué, dupliquant également la lourdeur de son implémentation. Dans ce cas, il est préférable d'utiliser une solution à deux niveaux. Il s'agit alors de séparer la notion de DDS et la notion de DPMap dans des objets différents. Le DPMap est alors réduit à un objet très léger qui ne contient pas d'informations sur le maillage, sa distribution, et sa connectivité, mais uniquement un stockage de données associées, par un moyen ou un autre, au DDS (par une indexation ou une référence par exemple). Dans ce cas, le maillage μ est implémenté par le DDS et la fonction σ est implémentée par m instanciations de composants DPMap, où m représente le nombre de quantités à simuler.

3.3 Applicateurs et opérations

Comme expliqué précédemment dans l'analyse BSP du programme parallèle d'une simulation scientifique, chaque super-étape de calcul, qui calcule un schéma numérique S, correspond à un code séquentiel qui sera demandé à l'utilisateur. Dans l'approche SIPSim, un code séquentiel de l'utilisateur est appelé une *opération*. Toutefois, une opération ne peut pas être appelée et appliquée directement dans le code utilisateur, comme une fonction séquentielle. En effet, une opération est appelée par le biais du composant SIPSim appelé *applicateur*. Un applicateur cache à l'utilisateur les communications nécessaires au bon déroulement du calcul de S. En effet, dans tout calcul stencil distribué, des communications sont nécessaires entre les processeurs pour obtenir le voisinage $N(x)$ dont ils n'ont pas la charge localement. Ces échanges d'informations sont donc nécessaires avant de procéder au calcul de S et l'applicateur va se charger de ces échanges. Un applicateur va donc tout d'abord effectuer de façon implicite des communications MPI, puis appliquer l'opération. Un applicateur est une procédure définie par

$$Applier : DPMap^* \times Operation \tag{3.1}$$

où $DPMap^*$ représente un ensemble d'instanciations d'objets DPMap, et donc un ensemble de quantités sur lesquelles appliquer le schéma numérique S. Précisons qu'un applicateur permet d'appliquer des calculs numériques sur un ensemble de quantités, par conséquent un applicateur ne s'intéresse pas directement au composant DDS mais aux instances des composants DPMap (eux même reliés par un moyen ou un autre au DDS sur lequel ils sont appliqués).

3.4 Interfaces de programmation

Dans le code d'une opération, un ou plusieurs schéma numériques S vont être calculés, ce qui implique de devoir manipuler des objets distribués de façon implicite. En effet, tout d'abord, le schéma numérique S doit être calculé pour l'ensemble des éléments du maillage, il est donc nécessaire de pouvoir naviguer dans les éléments du maillage distribué de façon implicite. De plus, comme indiqué dans l'équation (2.3), la fonction σ doit être manipulée dans le code séquentiel. L'instanciation de m DPMap permet d'implémenter la fonction σ, et le code de l'opération va donc devoir accéder aux valeurs des instances du composant DPMap. Pour finir, afin de calculer S il faut pouvoir accéder au voisinage $N(x)$ d'un élément $x \in E_{\mu_i}$. Par conséquent il est nécessaire de fournir un moyen simple d'accéder aux éléments du stencil.

Pour ces raisons, trois interfaces de programmation sont recommandées par l'approche SIPSim. Elles permettent de coder les opérations de façon séquentielle. Tout d'abord, en génie logiciel, un *itérateur* est un *patron de conception*, ou *design pattern*, qui permet de parcourir des éléments contenus dans une liste, ou un objet quelconque. Ce concept est très utilisé dans de multiples langages séquentiels comme C++, Java, C#, Python

etc., mais aussi, comme nous l'avons vu dans la section 2.4, dans des langages et bibliothèques parallèles tels que STAPL, PSTL etc. Dans l'approche SIPSim, ce concept est réintroduit. Il permet de naviguer dans une structure de données, en l'occurrence dans une instance DPMap et donc, de façon sous-jacente, dans une DDS. Un DPMap étant un objet distribué, un itérateur SIPSim est donc un itérateur qui cache le fait que chaque processeur ne navigue que dans une sous-partie de la structure. Un itérateur SIPSim permet à chaque processeur p_i de naviguer dans E_{μ_i}. Notons une nouvelle fois que E_{μ_i}, suivant le type de maillage et le type de résolution numérique, peut représenter, par exemple, les points ou les arêtes du maillage, ou encore le centre des mailles. Au moins deux types d'itérateurs sont nécessaires dans l'approche SIPSim et permettent l'efficacité de la solution. Tout d'abord un certain nombre d'itérateurs devront permettre de naviguer dans la bordure physique du maillage. Ces itérateurs permettront, comme dans le code séquentiel, de traiter les conditions limites de la simulation sans devoir ajouter de nombreuses conditions dans le code. Il sera ensuite nécessaire de fournir un ensemble d'itérateurs permettant de naviguer dans les autres éléments du maillage E_{μ_i}. La façon dont ces itérateurs parcourent les éléments du maillage n'a pas d'importance, du fait de l'absence de dépendance entre les éléments dans le schéma numérique (équation (2.3)). Le développeur de la solution SIPSim pourra alors implémenter les itérateurs et leurs mouvements de façon à optimiser les performances, optimiser l'utilisation du cache, favoriser la vectorisation etc.

Une fois les itérateurs mis en place, il sera nécessaire de proposer une deuxième interface permettant d'accéder aux données contenues dans les instances de DPMap pour cet itérateur. Un itérateur peut, par exemple, pointer sur un élément précis d'une instance de DPMap précise. Dans ce cas, il fonctionnera comme un itérateur de la STL C++. Un itérateur pourrait aussi être vu comme un index général de positionnement dans le maillage, et permettrait alors l'accès à toutes les instances DPMap pour l'index courant.

Enfin, une troisième interface de programmation sera nécessaire afin d'accéder aux valeurs voisines de l'itérateur courant, celles définies dans le stencil de la simulation. Cela pourra se faire, par exemple, au travers d'un ensemble de fonctions.

3.5 Vue utilisateur et vue réelle

Les quatre composants qui viennent d'être décrits permettent de proposer une vision générale et séquentielle du programme à l'utilisateur, alors que le véritable fonctionnement du programme est de type SPMD. La figure 3.1 illustre ces deux visions du programme final.

Ce que doit comprendre l'utilisateur au travers d'une solution SIPSim (à gauche de la figure), c'est (1) qu'il doit écrire ses programmes séquentiels en une ou plusieurs opérations, en utilisant les interfaces de programmation à disposition, (2) qu'il doit ensuite appliquer ces opérations au travers d'applicateurs, et enfin (3) que ces opérations seront appliquées sur un ensemble de données de type DPMap, elles-mêmes appliquées sur un

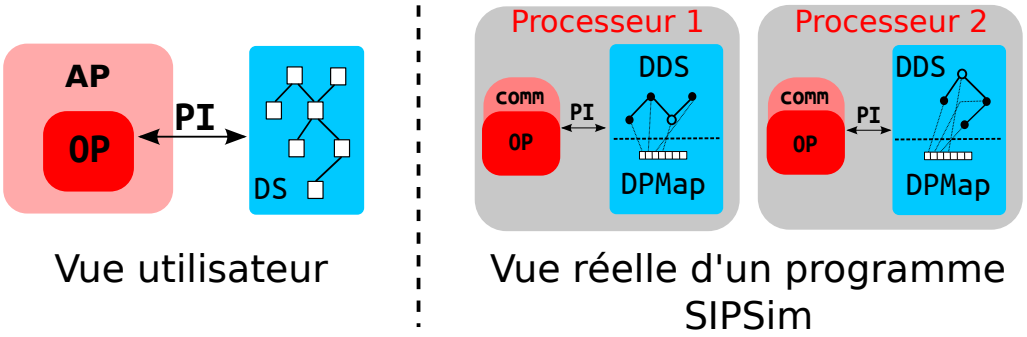


FIGURE 3.1 – *Vue utilisateur (à gauche) et vue réelle (à droite) d'un programme SIPSim.*

DDS représentant le maillage du problème. Ce qui se passe réellement dans un programme implémenté avec une solution SIPSim, c'est que l'objet DDS est distribué sur plusieurs processeurs, tout comme les données DPMap qui lui sont appliquées, et qu'un applicateur va procéder aux communications nécessaires entre les processeurs avant l'appel à l'opération de l'utilisateur.

Cette double vue, proposée par le modèle SIPSim, permet d'obtenir un modèle de programmation parallèle totalement implicite pour des simulations basées sur des maillages.

3.6 Type de programmation

L'approche SIPSim a donc pour but de proposer une façon systématique d'implémenter des solutions de parallélisme implicite pour les simulations scientifiques. L'approche se base sur quatre composants pour proposer une façon de coder proche du séquentiel, comme elle a été présentée dans l'algorithme 1. Nous avons décrit dans ce chapitre les quatre composants de l'approche, nous allons maintenant observer comment est typiquement utilisée une solution implémentée suivant l'approche SIPSim.

L'algorithme 4 donne un exemple type de fonction principale dans un programme écrit avec une solution SIPSim. Dans ce pseudo-code, il est tout d'abord nécessaire de créer un maillage distribué à l'aide de l'objet DDS, puis d'y appliquer des quantités à simuler, par l'instanciation de l'objet ou des objets DPMap disponibles. L'initialisation locale des instances de DPMap sont effectuées en utilisant les interfaces de programmations décrites précédemment : les itérateurs, les accesseurs. La principale différence avec le véritable programme séquentiel est l'appel à des applicateurs dans la boucle principale. En effet, comme nous l'avons vu, une opération, qui permet de calculer un schéma numérique pour une itération donnée, ne peut être directement appliquée puisqu'elle nécessite un certain nombre d'échanges de valeurs entre les processeurs. Pour cette raison, les opérations sont appelées par le biais d'un applicateur. Dans cet exemple, deux applicateurs sont appelés à chaque itération de temps.

Algorithme 4 : Exemple de fonction principale avec l'approche SIPSim

```
Création du maillage distribué DDS
Instanciation des DPMap sur le DDS
Initialisations des instances DPmap et des paramètres
Définition du pas de temps, commun à tous les processeurs : Δt
Définition du temps maximal, commun à tous les processeurs : tmax
tant que t < tmax faire
    applicateur(DPMap*,opération1)
    applicateur(DPMap*,opération2)
    t=t+Δt
fin
```

L'algorithme 5, de son côté, représente un exemple de pseudo-code d'opération. Dans cet exemple, l'ensemble des éléments de la bordure physique est tout d'abord traité, puis ensuite les autres éléments du maillage. Pour cette raison deux boucles *for* sont implémentées, une première qui utilise les itérateurs sur la bordure physique, et une deuxième qui utilise les itérateurs sur les autres éléments du maillage. Pour "obtenir les valeurs", l'itérateur est utilisé sur les instances de DPMap. De même afin d'"obtenir les valeurs voisines", l'itérateur est utilisé sur les fonctions permettant d'accéder aux valeurs du stencil. Une fois ces valeurs obtenues, il est possible de calculer le schéma numérique.

Algorithme 5 : Exemple d'opération SIPSim

```
itB := point de démarrage de la bordure physique
enditB := point de fin de la bordure physique
tant que itB≤enditB faire
    Obtenir les valeurs à itB : σ(itB, t − 1)
    Obtenir les valeurs voisines à itB : σ(N(itB), t − 1)
    Calcul de S_b(itB, t)
    itB++
fin
it= point de démarrage des autres éléments
endit := point de fin des autres éléments
tant que it≤endit faire
    Obtenir les valeurs à it : σ(it, t − 1)
    Obtenir les valeurs voisines à it : σ(N(it), t − 1)
    Calcul de S(it, t)
    it++
fin
```

Nous pouvons observer dans ces deux pseudo-codes que la programmation en utilisant une solution SIPSim est très proche d'une programmation séquentielle. Il est toutefois

nécessaire d'utiliser les quatre composants de l'approche : DDS, DPMap, AP et PI. De nouvelles notions sont donc présentes pour l'utilisateur de la solution. Toutefois, ces notions sont très proches des notions de programmation objet classique et donc peu complexes à comprendre. Enfin, le but recherché est atteint puisque le code parallèle est complètement implicite pour l'utilisateur.

3.7 Conclusion

Dans ce troisième chapitre, nous avons introduit la méthode SIPSim, qui propose une approche systématique, grâce à quatre composants, pour implémenter des solutions de parallélisme implicite pour la résolution d'EDP basées sur des maillages. Cette approche peut donc être considérée comme un *modèle de programmation implicite*. L'ensemble des travaux présentés dans la suite de cette thèse seront basés sur ce modèle de programmation implicite.

La mise en place d'une solution SIPSim est libre. En effet, aucune indication n'est donnée sur la façon d'implémenter la solution, mais juste sur les composants nécessaires au modèle et à son abstraction. Il est donc possible d'implémenter la solution de diverses façons, comme une langage de domaine spécifique, une bibliothèque, une framework etc. Dans le reste de cette thèse nous allons présenter une implémentation possible du modèle SIPSim. Cette implémentation se nomme *SkelGIS*, il s'agit d'une bibliothèque C++ uniquement composée de fichier d'en-tête, on dit aussi *header-only*, ou *h-only*. Il s'agit donc d'une solution légère, comme cela se fait souvent dans les bibliothèques de squelettes algorithmiques.

SkelGIS pour des maillages réguliers à deux dimensions

4

Sommaire

Dans le chapitre 3, la méthode *SIPSim*, permettant de systématiser la création de solutions de parallélisme implicite pour les simulations scientifiques basées sur des maillages, a été introduite. Cette thèse met en œuvre l'implémentation et l'évaluation de la méthode SIPSim sous forme d'une bibliothèque nommée *SkelGIS*. Dans ce chapitre nous allons tout d'abord détailler l'implémentation de SkelGIS dans le cas des simulations sur des maillages cartésiens à deux dimensions. S'en suivront deux cas d'application réels. Le premier consiste à résoudre l'équation de la chaleur. Cette résolution sera détaillée en utilisant la bibliothèque SkelGIS, mais également en utilisant MPI (Message Passing Interface). Des comparaisons de performances et d'effort de programmation, de ces simulations, seront analysées. Le second cas d'application, plus complexe, concerne la résolution des équations de Saint-Venant dans le cadre d'une simulation d'écoulement des eaux. L'implémentation de cette simulation en utilisant SkelGIS sera comparée avec une version équivalente développée avec la bibliothèque MPI.

4.1 SIPSim pour les maillages réguliers à deux dimensions

4.1.1 Structure de données distribuée

Comme expliqué dans la partie 3.1, le premier composant de la méthode SIPSim est la *structure de données distribuée*, aussi appelée *DDS*. Ce composant permet de représenter le maillage et sa connectivité. Il est aussi le composant principal de la solution, sur lequel tout le reste de l'implémentation repose. Dans SkelGIS, et dans le cadre des maillages cartésiens à deux dimensions, l'implémentation du DDS est assez simple. Le DDS est implémenté au travers d'une classe C++ qui porte le nom de *DMatrix*. Ce nom est issu du fait qu'il semble naturel de représenter des données sur un maillage régulier à deux dimensions par des matrices.

La connectivité d'un maillage régulier est régulière. Cette propriété paraît évidente mais elle met en avant qu'il n'est pas nécessaire, dans une DDS représentant un maillage régulier, de stocker la connectivité du maillage puisque celle-ci est la même pour tous les éléments qui le constitue. Dans la version actuelle de SkelGIS, seuls deux choix de connectivités sont possibles pour un maillage régulier à deux dimensions. Le premier est appelé *étoile*, ou *star*, et est représenté à gauche de la figure 4.1. Dans ce cas la connectivité nécessaire pour un calcul sur le maillage, que l'on pourrait aussi appeler le voisinage nécessaire pour le calcul des éléments du maillage, est constituée des quatre directions principales autour d'un élément. Le deuxième, appelé *boîte*, ou *box*, est représenté à droite de la figure 4.1. Dans ce cas, la connectivité nécessaire pour un calcul sur le maillage est constituée des huit directions principales autour d'un élément. Le choix de cette connectivité sera le même pour tous les éléments de l'instance *DMatrix*. Pour cette raison, les *DMatrix* de SkelGIS ne stockent pas la connectivité du maillage régulier. Seul le type de connectivité est nécessaire pour retrouver ensuite le voisinage des éléments. Étant donné que l'objet *DMatrix* est léger, il a été choisi de stocker directement les données appliquées

au maillage dans cet objet. Ainsi, l'objet *DMatrix* embarque directement le composant DPMap (*application de données*) de la méthode SIPSim. Dans un code de simulation SkelGIS, il y aura donc autant d'instanciations de l'objet *DMatrix* que de quantités à simuler dans les équations. Notons que des instanciations de DMatrix supplémentaires peuvent permettre, par exemple, de conserver des résultats temporaires. Afin de rendre

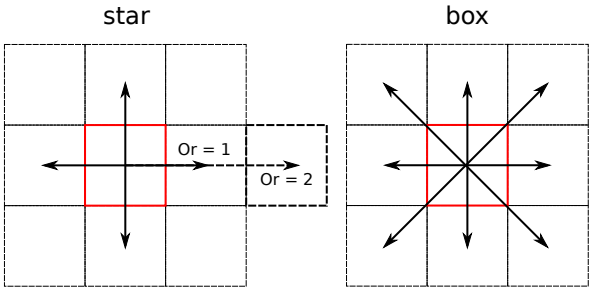


FIGURE 4.1 – *Deux types de connectivités pour les DMatrix de SkelGIS*

léger et efficace l'objet *DMatrix* les matrices sont stockées dans des tableaux à une dimension, mais ce choix n'a pas été fait que par soucis de simplicité et de légèreté. Dans un code de simulation scientifique basée sur un maillage, à chaque itération de temps sont calculés des schémas numériques explicites du type 2.3. Ce type de schéma numérique implique que le calcul du maillage à l'itération de temps $t+1$ ne dépend que du maillage à l'itération précédente t. Cela signifie qu'il n'y a pas de dépendances entre les éléments du maillage à un même instant t. Pour cette raison, ce type de simulation est dite "non ordonnée". Dans une simulation non-ordonnée, l'ordre de parcours des éléments du maillage n'a pas d'importance et le résultat produit sera le même si l'ordre de parcours est modifié. Il est alors préférable pour l'efficacité de la solution de se déplacer de façon contiguë en mémoire pour effectuer les calculs. Ainsi l'utilisation des lignes de cache de la machine sera optimisée. En effet, sur une machine contenant des lignes de cache de taille c, et étant donné un tableau contenant n éléments de taille k lus de façon contiguë, le nombre de défauts de cache est au minimum égal à $n \times k/c$. Il s'agit alors de défauts de cache capacitifs. Les performances d'un tel cas seront meilleures que dans le cas de défauts de cache conflictuels. Pour cette raison un tableau à une dimension est plus efficace pour représenter une matrice et c'est ce choix d'implémentation qui a été fait pour l'objet *DMatrix* dans SkelGIS.

Comme expliqué précédemment, la méthode SIPSim génère des programmes parallèles de type SPMD (Simple Program Multiple Data) pour des architectures à mémoire distribuée. Dans ce type de programme, un facteur majeur d'efficacité est d'effectuer une bonne décomposition du domaine en espace. Lorsqu'une bonne décomposition de domaine est effectuée, chaque processeur possède la même quantité de travail à effectuer tout en minimisant le nombre de communications nécessaires entre les processeurs. Dans le cadre d'un maillage cartésien à deux dimensions, comme nous l'avons vu dans la section 2.3.3.1, une décomposition de domaine n'est pas difficile. Les deux partitionnements

rectilignes sont les plus utilisés pour des maillages denses cartésiens, et sont représentés dans la figure 4.2.

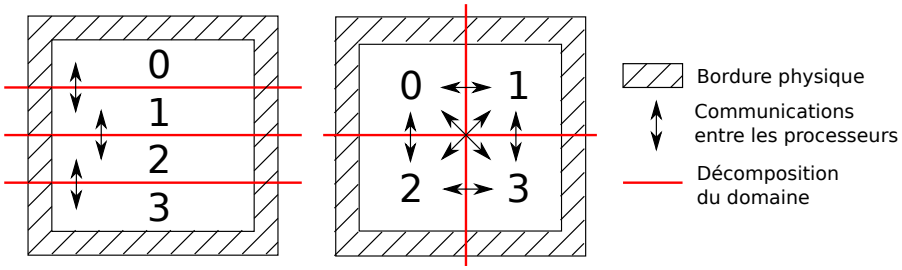


FIGURE 4.2 – *Décomposition d'un domaine à deux dimensions.*

Les deux décompositions représentées dans la figure 4.2 ont été implémentées dans SkelGIS. Les performances observées sont très proches et les deux solutions apportent des avantages.

La décomposition rectiligne 1D, ou en blocs-lignes, représentée à gauche de la figure 4.2, est intéressante pour équilibrer la distribution des bordures physiques de la simulation. Il a déjà été évoqué dans ce manuscrit la notion de bordure physique d'une simulation scientifique. Une simulation scientifique est, en effet, contrainte par un domaine en espace borné alors que l'espace du phénomène réel simulé n'est pas restreint et peut être impacté par d'autres objets ou phénomènes autour de lui. Pour cette raison, dans une simulation scientifique, il est fréquent d'ajouter des conditions limites aux équations pour simuler un comportement spécifique au bord du domaine. Les éléments concernés font partie de la bordure physique de la simulation. Ces éléments ajoutent des contraintes dans le partitionnement du travail pour les processeurs. Ils ont la particularité d'être globaux, ce qui signifie que tous les processeurs n'auront pas la même partie de bordure physique à gérer suivant la décomposition choisie. Cette idée est illustrée dans la figure 4.2 par les parties hachurées. Dans le cas d'une décomposition 2D, ou en blocs (à droite de la figure), la bordure physique peut fortement déséquilibrer la décomposition. En effet, les processeurs dans les coins devront calculer des bordures physiques sur la moitié de leur côtés de sous-domaines, les processeurs sur les bords auront eux un côté entier de leur sous-domaine concerné par la bordure physique, et enfin les processeurs à l'intérieur du domaine n'auront aucune bordure physique à calculer. Avec une décomposition en lignes (à gauche de la figure), le problème est moindre. En effet, à l'exception des processeurs 0 et $p-1$, où p représente le nombre de processeurs, les processeurs auront la même quantité de bordure physique à calculer sur les côtés de leurs sous-domaines ce qui équilibre naturellement la décomposition. Notons, de plus, que dans le cas d'une décomposition en lignes, la distribution de la bordure physique aux processeurs sera plus simple que dans le cas d'une décomposition en blocs.

La décomposition en ligne apporte également un avantage quant à la minimisation du nombre de processeurs avec lesquels il est nécessaire de communiquer, et quant à la simplicité du code. La figure 4.2 illustre parfaitement ce point par les flèches, qui

représentent les communications à effectuer à chaque itération de temps pour obtenir le voisinage nécessaire au calcul du sous-domaine de chaque processeur. Dans le pire des cas, si une connectivité de type *box* est nécessaire, alors la décomposition en blocs nécessitera des échanges de données avec trois processeurs dans les coins du domaine, avec cinq processeurs sur les bords du domaine et avec huit processeurs à l'intérieur du domaine. Une décomposition en lignes, à l'inverse, et dans tous les cas de connectivité, ne nécessitera que des communications avec un processeur pour les processeurs 0 et $p-1$, et avec deux processeurs pour tous les autres. Il est en revanche évident que le volume d'une seule communication, dans le cas d'une décomposition en lignes, sera plus important que celles d'une décomposition en blocs. Dans le cas d'un domaine très grand et de très gros volumes de données à traiter, la taille des messages peut devenir un problème. Toutefois, étant donnée la vitesse des réseaux actuels, il est souvent plus coûteux d'initier des communications que de transférer les données en elles-mêmes. De plus, des optimisations de recouvrement des communications avec les calculs permettent d'effectuer les transferts de données sur le réseau en limitant l'impact sur les performances du programme.

Les résultats présentés dans la suite de ce chapitre ont été obtenus avec une décomposition rectiligne 1D. La décomposition 2D en blocs offrait toutefois des performances quasiment équivalentes.

4.1.2 Applicateurs et opérations

Une fois qu'un ensemble d'instances de *DMatrix* a été créé, il est nécessaire d'y appliquer des calculs. Cela se fera à travers l'utilisation d'applicateurs, comme il a été présenté dans le chapitre 3. Pour des maillages cartésiens à deux dimensions, un applicateur est une fonction qui applique, comme son nom l'indique, une fonction séquentielle codée par l'utilisateur (une opération) sur des instances de *DMatrix*. L'ensemble des communications MPI nécessaires aux calculs est caché à l'utilisateur dans l'appel de l'applicateur. Pour les *DMatrix* il existe quatre applicateurs dans le prototype actuel de SkelGIS, dont voici les définitions :

$$ApplyUnary : F_{un} \times DMatrix \longrightarrow DMatrix \tag{4.1}$$

$$F_{un} = \{f : DMatrix \longrightarrow DMatrix\} \tag{4.2}$$

$$ApplyBinary : F_{bin} \times DMatrix \times DMatrix \longrightarrow DMatrix \tag{4.3}$$

$$F_{bin} = \{f : DMatrix \times DMatrix \longrightarrow DMatrix\} \tag{4.4}$$

$$ApplyList : F_{list} \times list(DMatrix) \longrightarrow list(DMatrix) \tag{4.5}$$

$$F_{list} = \{f : list(DMatrix) \longrightarrow list(DMatrix)\} \tag{4.6}$$

$$ApplyReduction : F_{reduc} \times DMatrix \longrightarrow E \tag{4.7}$$

$$F_{reduc} = \{f : DMatrix \longrightarrow E\} \tag{4.8}$$

Le premier, le plus simple, nommé *ApplyUnary* (4.1) permet d'appliquer une *opération* de type (4.2) à une instance de *DMatrix* en entrée, et retourne une nouvelle instance

de *DMatrix* en sortie. Le deuxième, *ApplyBinary* (4.3), permet d'appliquer une *opération* de type (4.4) à un couple d'instance de *DMatrix* en entrée, et retourne une nouvelle instance de *DMatrix* en sortie. Une généralisation de ces deux applicateurs est fournie avec l'applicateur *ApplyList* (4.5). Il permet d'appliquer une *opération* de type (4.6) à un ensemble d'instances de *DMatrix* en entrée, et retourne un ensemble de *DMatrix* en sortie. Enfin, le dernier applicateur est l'applicateur *ApplyReduction* (4.7), qui permet d'appliquer une *opération* de type (4.8) à une instance de *DMatrix* en entrée, et retourne un unique élément en sortie. Comme il l'a déjà été évoqué, les applicateurs sont proches des notions de squelettes algorithmiques, et ici il est évident qu'un parallèle peut être fait entre ces applicateurs et les squelettes algorithmiques *map*, *zip* et *reduce*. Ils n'en sont pas si éloignés, mais les applicateurs SIPSim ne sont pas d'un niveau d'abstraction aussi haut que les squelettes algorithmiques. En effet, ces squelettes cachent le parcours des éléments de la structure de données distribuée, alors que SkelGIS au contraire souhaite rester proche d'une programmation séquentielle, en conservant la vision générale de l'utilisateur sur le programme et sur les structures de données. De plus, grâce au dernier composant de la méthode SIPSim, les interfaces de programmation, l'utilisateur va pouvoir accéder au voisinage des éléments du maillage ce qui n'est pas possible avec les squelettes algorithmiques *map*, *zip* et *reduce* évoqués précédemment. Les squelettes algorithmiques sont d'un niveau d'abstraction plus élevé, mais sont aussi plus généralistes que l'approche SIPSim. SIPSim est une solution spécifique aux simulations scientifiques et peut, par exemple, gérer la particularité des bordures physiques.

4.1.3 Interfaces de programmation

Le dernier point à résoudre est le moyen de coder des *opérations*. Pour ce faire, la méthode SIPSim propose un dernier composant qui comprend trois types d'interfaces de programmation, les *itérateurs* et les outils d'accès aux éléments et aux éléments du stencil.

4.1.3.1 Itérateurs

Les *itérateurs* permettent de se déplacer dans les *DMatrix*, d'accéder et de modifier leur données. Dans le cadre de maillages cartésiens à deux dimensions, huit itérateurs ont été implémentés. Six d'entre eux sont plus importants et vont permettre d'obtenir de meilleures performances s'ils sont préférés aux autres. Les deux itérateurs restants permettent de naviguer différemment dans les *DMatrix* mais seront moins efficaces.

Tout d'abord, deux *itérateurs* permettent de naviguer dans des *DMatrix* de façon contiguë en mémoire. Pour ces itérateurs, seul l'opérateur ++ est disponible ce qui limite la liberté de déplacement mais garantit une optimisation dans l'utilisation de la mémoire cache. Le premier, nommé *iterator_cont*, permet de naviguer du premier au dernier élément d'une instance de *DMatrix*. Le deuxième est l'équivalent dans le sens inverse. *iterator_rev* permet de se déplacer du dernier au premier élément d'une instance de *DMatrix*.

Ensuite, quatre itérateurs spécifiques permettent de naviguer parmi les éléments de la bordure physique. Les itérateurs *iterator_phb_up*, *iterator_phb_down*, *iterator_phb_right*, *iterator_phb_left* permettent respectivement d'explorer les éléments de la partie haute, basse, droite et gauche de la bordure physique. Notons que ces quatre itérateurs sont importants car chaque côté du domaine a potentiellement des conditions limites différentes. Il est très important de différencier les éléments du domaine des éléments de la bordure physique. En effet, cette technique permet d'éviter des conditions coûteuses dans le code pour connaître le type de l'élément en cours d'exploration.

Par la suite, l'implémentation d'un itérateur plus standard a été effectuée et permet d'accéder aux opérateurs $++, --, +=, -=$ du début à la fin de l'instance du *DMatrix*. Enfin un dernier itérateur permet de définir des mouvements réguliers que l'on souhaite effectuer dans l'instance du *DMatrix*, en indiquant un nombre d'éléments contigus à lire et un saut à effectuer une fois ces éléments lus, et avant de lire les suivants. Ce dernier itérateur est illustré dans la figure 4.3. Ces deux derniers itérateurs permettent donc de

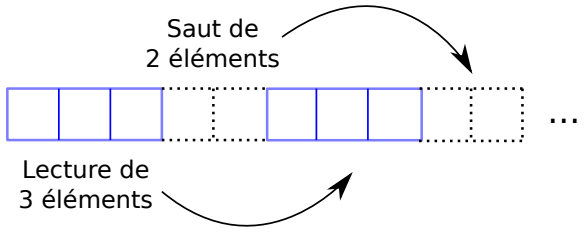


FIGURE 4.3 – *Exemple d'itérateur permettant de se déplacer dans trois éléments contigus puis d'effectuer un saut de deux éléments avant la lecture contiguë suivante.*

se déplacer de façon non contiguë en mémoire ce qui peut nuire à la performance du programme final. Toutefois, il peut être utile de faire ce type de déplacements.

Dans l'implémentation actuelle de SkelGIS, un itérateur n'est pas un pointeur, contrairement à la STL, par exemple. Nous avons, en effet, fait le choix d'associer le terme d'itérateur à un index, ce qui permet avec un unique itérateur positionné dans le maillage de pouvoir accéder aux différentes quantités de l'élément du maillage.

4.1.3.2 Accesseurs

Lorsqu'un itérateur *it* est obtenu, et correspond à un élément du maillage, il est possible d'accéder et de modifier les données sur cet élément. Les données sur l'élément du maillage sont contenues dans les instances de l'objet *DMatrix*. Il est ainsi possible pour une instance m d'obtenir sa valeur en effectuant $m[it]$ et de modifier sa valeur par val en effectuant $m[it] = val$.

4.1.3.3 Voisinages

Comme décrit précédemment, il existe deux types de connectivité possibles dans l'objet *DMatrix* de SkelGIS : la connectivité *star* et la connectivité *box* représentées dans

la figure 4.1. Le voisinage d'un élément du maillage est, par définition, lié à la connectivité du maillage. Il est en effet évident qu'un élément ne peut être défini comme voisin d'un autre s'ils ne sont aucunement reliés dans le maillage. Le type de connectivité donne donc une indication sur le voisinage nécessaire pour le calcul d'un élément du maillage, mais une information importante reste manquante : à quelle distance de voisinage avons-nous accès ? Dans une simulation scientifique lorsque des schémas numériques sont appliqués, les solutions peuvent être plus ou moins précises suivant la distance de voisinage auquel on accède dans le calcul. Cette information est appelée l'*ordre* de la simulation (cette notion a été introduite dans l'état de l'art 2.2). Lorsque le type de connectivité et l'ordre de la simulation sont connus, il est possible de déterminer le voisinage des éléments d'un maillage et de définir les interfaces nécessaires pour y accéder. Par exemple, si l'ordre d'une simulation est égal à 1 et que le type de connectivité est *star* alors il est possible d'accéder aux quatre voisins directs d'un élément. Si l'ordre est égal à 2, il est alors possible d'accéder aux éléments dans les quatre directions à une distance de deux mailles etc (figure 4.1). Ainsi, suivant la connectivité et l'ordre précisé par l'utilisateur de SkelGIS différentes interfaces de programmation pour accéder au voisinage ont été implémentées. Elles sont de trois types. Tout d'abord il est possible d'obtenir, pour une instance de *DMatrix*, la ou les valeurs voisines (suivant l'ordre) dans une direction donnée. Par exemple, si m est une instance de DMatrix, et it un itérateur, on peut accéder aux voisinages par les méthodes $m.getRight(it, ordre)$, $m.getDown(it, ordre)$ etc. Il est ensuite possible d'obtenir, pour une instance de *DMatrix*, l'ensemble des valeurs voisines verticalement ou horizontalement, avec $m.getX(it, ordre)$, $m.getY(it, ordre)$. Enfin, il est possible d'obtenir une liste contenant toutes les valeurs voisines, pour une instance de *DMatrix*, avec $m.getAll(it, ordre)$.

4.1.4 Spécialisation partielle de template

Après les premiers prototypes de la méthode SIPSim, il s'est avéré que certaines conditions dans le code étaient très coûteuses en temps d'exécution car exécutées un très grand nombre de fois. En effet, étant donnée une simulation de t itérations en temps, et un maillage cartésien à deux dimensions contenant n éléments, certaines conditions peuvent être exécutées $t \times n$ fois, ce qui peut impacter de façon importante les performances. Nous nous sommes alors attachés à éviter au maximum les conditions dans le code SkelGIS. Le premier type de condition qu'il est possible d'éviter a déjà été décrit. Il s'agit du test sur le type d'un élément pointé par un itérateur. Nous avons assez vite pris la décision de séparer les éléments de la bordure physique des autres éléments et avons proposé des itérateurs spécifiques pour naviguer dans les éléments spécifiques sans le moindre besoin de conditions dans le code. Cependant d'autres conditions étaient très coûteuses : les tests sur le type de connectivité choisi par l'utilisateur et les tests sur l'ordre de la simulation. Ces conditions ne sont pas contrôlées par l'utilisateur mais cachées au sein de la bibliothèque SkelGIS puisqu'elles permettent de fournir les interfaces adaptées au code utilisateur. La première façon de gérer ces différents cas aurait été de faire appel au système d'héritage virtuel du C++. Ce système, très esthétique, aurait permis de définir une classe mère de l'objet *DMatrix* permettant de définir les attributs et méthodes

communes, ainsi que les méthodes virtuelles pures pour les comportements spécialisés. Puis, à cette classe mère seraient venues s'ajouter des classes filles afin de spécialiser les différents comportements dans les méthodes virtuelles, suivant la connectivité du maillage et suivant l'ordre de simulation. Cependant le mécanisme d'héritage virtuel ajoute un surcoût à l'exécution, qui peut devenir très important lorsque les appels aux méthodes virtuelles se font de façon fréquente et pour un calcul peu coûteux. En effet, dans le cas d'une méthode virtuelle, une table d'héritage est créée à la compilation, et un pointeur vers cette table d'héritage est également créé. Ce pointeur permettra, à l'exécution, le choix du bon appel de méthode. Cette méthode engendre donc une indirection supplémentaire à chaque appel lors de l'exécution. Dans notre cas l'accès au voisinage d'un point peut se faire plus de $nbI \times nbE$ fois, où nbI représente le nombre d'itérations de temps de la simulation, et nbE le nombre de points à calculer dans le maillage. Une autre solution aurait été de définir plusieurs objets pour chaque type de *DMatrix* possible. Cette fois la solution aurait été efficace mais complexe pour l'utilisateur puisqu'il y aurait eu autant d'objet à sa disposition que de possibilités de connectivités et d'ordre.

Nous nous sommes tournés vers une autre solution appelée la *spécialisation partielle de template*. Ce mécanisme connu du C++ permet de produire un code générique, tout en obtenant de bonnes performances, en utilisant les propriétés du compilateur C++ dans le cas des templates. A la compilation d'un programme C++, pour chaque classe template instanciée dans le code, la bonne classe à instancier est déterminée suivant son nom et les paramètres template spécifiés. Une copie du code correspondant est ensuite effectuée, pour chaque instance, dans le code compilé. Il est de plus possible en C++ de partiellement ou totalement spécifier les paramètres d'une classe template pour en proposer une implémentation différente suivant les valeurs des paramètres. Avec ces deux mécanismes du C++, il est possible de spécifier le code, et cette spécification est déterminée à la compilation et non à l'exécution du code. Avec une bonne identification des informations à passer en paramètres de template, il est possible d'éviter des conditions coûteuses dans le code, tout comme dans un mécanisme d'héritage, mais sans perte de performances à cause des indirections.

La figure 4.4 donne la définition de l'objet *DMatrix* et ses trois spécialisations partielles de template.

```
template<class T, int Or, bool box=false> struct DMatrix
template<class T> struct DMatrix<T,0>
template<class T, int Or> struct DMatrix<T, Or, true>
template<class T> struct DMatrix<T, 0, true>
```

FIGURE 4.4 – *Spécialisation partielle de template pour l'objet DMatrix : T est le type de donnée à stocker dans l'instance, Or est l'ordre de la simulation, et box est le type de connectivité souhaitée. Ce paramètre a une valeur par défaut à false (star est le choix par défaut).*

Trois paramètres de template sont définis pour l'objet *DMatrix*. Le premier paramètre, *T*, indique le type de données qui va être stocké dans l'instance de l'objet. Le

paramètre *Or* indique ensuite l'ordre nécessaire pour l'instance de l'objet *DMatrix*. Ce paramètre peut sembler inutile pour l'objet *DMatrix* en lui-même, puisque l'ordre est une indication qui concerne la simulation et non chaque quantité simulée. Toutefois, il est important de demander cette information au niveau de l'objet *DMatrix* pour rendre la solution plus efficace. En effet, dans une simulation, toutes les quantités à simuler sont nécessaires au calcul du ou des schémas numériques de type (2.3), toutefois, toutes les quantités à simuler ne participent pas aux calculs faisant intervenir $N(x)$. Demander l'ordre pour chaque instance de l'objet *DMatrix* permet donc d'éviter des communications inutiles lorsque l'instance n'a pas besoin de voisinages. Enfin, le troisième paramètre de la classe template, *box*, est un booléen avec une valeur par défaut à *false*. Ce paramètre indique la connectivité du maillage qui va être définie en instanciant l'objet *DMatrix*. En effet, rappelons ici que l'objet *DMatrix* a la particularité de regrouper deux composants de la méthode *SIPSim*, la structure de données distribuée (DDS), et l'application de données sur cette structure de données (DPMap). L'instanciation d'un objet *DMatrix* correspond donc bien à la définition d'un maillage et il n'est pas impossible qu'une simulation complexe instancie différents types de connectivités pour ses différentes quantités à simuler.

Trois spécialisations partielles des paramètres de cette classe sont proposées et données dans la figure 4.4. Tout d'abord, en conservant la valeur par défaut du paramètre *box*, le paramètre *Or* est spécialisé avec la valeur 0. Cette spécialisation permet d'indiquer qu'une quantité utilisée dans la simulation ne participe pas aux calculs faisant intervenir $N(x)$. Autrement dit, ce type de quantité est utilisé localement, sans notion de voisinage. Cette spécialisation est très importante pour les performances de la solution puisqu'alors aucune communication MPI ne sera nécessaire. La deuxième spécialisation fixe le paramètre *box* avec une valeur à *true*. Cette spécialisation active donc la connectivité *box*. Dans cette spécialisation, les interfaces de voisinage proposées à l'utilisateur seront différentes. Enfin la troisième spécialisation fixe le paramètre *Or* à 0 et le paramètre *box* à *true*, ce qui combine les deux spécialisations précédemment décrites. Chaque spécialisation de l'objet *DMatrix* propose une implémentation de la classe qui lui est propre, ce qui alourdit considérablement le code de la bibliothèque (mais pas le code utilisateur). Cependant, cette solution est très intéressante puisque l'utilisateur ne manipule qu'une unique classe. De plus, les performances obtenues sont également très intéressantes puisque, dans le code, les conditions concernant l'ensemble de ces paramètres disparaissent. Enfin, le choix de la bonne implémentation de classe est effectué à la compilation et non à l'exécution ce qui rend cette solution efficace.

4.2 Résolution numérique de l'équation de la chaleur

4.2.1 Équation et résolution numérique

L'équation de la chaleur à deux dimensions est définie par

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

où $u(x, y, t)$ représente la température au point (x, y) et à l'itération de temps t.

Le schéma explicite des différences finies est le schéma le plus simple pour résoudre l'équation de la chaleur. Il consiste en une discrétisation du domaine en espace avec le maillage $\{x_i, y_j\}_{i,j}$, avec $x_i = i\Delta x$ et $y_j = j\Delta y$. $\Delta x = x_{i+1} - x_i$ et $\Delta y = y_{i+1} - y_i$ sont les intervalles en espace suivant les deux dimensions. Soit Δt l'intervalle de temps entre chaque itération, supposons qu'à un temps $t_n = n\Delta t$, la valeur $u^n_{i,j} = u(x_i, y_j, t_n)$ est connue pour chaque élément du maillage. Alors, en utilisant le développement polynomial de Taylor, la solution à l'instant t_{n+1} est donnée par le schéma suivant :

$$\frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t} = \frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta x^2} + \frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta y^2}.$$

En supposant que $\Delta x = \Delta y$, alors le schéma devient :

$$u^{n+1}_{i,j} = (1 - 4\lambda)u^n_{i,j} + \lambda(u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1}), \tag{4.9}$$

où $\lambda := \frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$ garantit la stabilité du schéma numérique.

4.2.2 Parallélisation avec SkelGIS

Le schéma numérique de l'équation (2.3) offre toutes les informations nécessaires pour coder la simulation en utilisant SkelGIS. Tout d'abord, seule la quantité u nécessite d'être manipulée dans le schéma. Ensuite, comme le calcul pour l'itération de temps $n+1$ dépend des résultats de l'itération de temps n, deux instanciations de l'objet *DMatrix* sont nécessaires pour stocker successivement les données d'entrée et de sortie. De plus, le schéma nous donne l'indication qu'une connectivité de type *star* est nécessaire pour chaque calcul. En effet, les éléments $(i + 1, j)$ (droit), $(i - 1, j)$ (gauche), $(i, j + 1)$ (bas) et $(i, j - 1)$ (haut) sont utilisés par le schéma. L'ordre de la simulation est égal à 1 car aucun élément aux positions $i \pm 2$ ou $j \pm 2$ ne sont nécessaires. Enfin, il y a un unique schéma à appliquer à chaque itération de temps, ce qui nous indique qu'un unique appel à un applicateur sera nécessaire pour cette résolution.

La figure 4.5 donne le code de la fonction *main* du programme de résolution de l'équation de la chaleur en utilisant SkelGIS. Deux instanciations de l'objet *DMatrix* sont tout d'abord effectuées (lignes 12 et 14). Par la suite, la boucle en temps est initialisée (ligne 15). A chaque itération de temps, l'applicateur est appelé avec l'opération *laplacien* (ligne 17) pour résoudre le schéma. Enfin, les *DMatrix* d'entrée et de sortie sont échangées pour l'itération en temps suivante (lignes 18 à 20). Notons ici que la création de $m3$ à la ligne 18 ne fait que copier l'adresse du pointeur qui est caché derrière l'objet m. Cette étape n'est donc pas coûteuse en temps d'exécution. Il reste toutefois quelques détails non précisés dans la figure 4.5. Les instructions *INITSKELGIS* et *ENDSKELGIS*, tout d'abord, permettent d'initialiser de façon implicite la bibliothèque MPI et certaines variables utiles à SkelGIS. La classe *HEADER* permet simplement de définir l'en-tête d'un maillage cartésien à deux dimensions suivant une largeur et une hauteur (width et height) et suivant une coordonnée en haut à gauche du maillage (x et y). Il est également

```
1  #include "skelgis/skelgis.hpp"
2  using namespace skelgis;
3
4  int main(int argc, char** argv)
5  {
6    INITSKELGIS;
7    HEADER head;
8    head.x=0;head.y=0;
9    head.width=100;head.height=100;
10   head.spacing=1;head.nodata=-9999;
11
12   DMatrix<double,1> m(head,0);
13   m.setGlobalMiddleValue(1);
14   DMatrix<double,1> m2(head,0);
15   for(int i=0;i<100;i++)
16     {
17       ApplyUnary<double,1,double,1>::apply(laplacien,m,m2);
18       DMatrix<double,1> m3(m);
19       m = m2;
20       m2 = m3;
21     }
22   ENDSKELGIS;
23 }
```

FIGURE 4.5 – *Fonction main du programme de résolution de l'équation de la chaleur avec SkelGIS.*

possible de préciser un nombre flottant représentant la hauteur et la largeur d'une maille, et de pouvoir identifier une valeur qui représente une maille sans donnée. Son nom est historiquement conservé de l'en-tête des fichiers représentant le terrain dans les *SIG* (Système d'Information Géographiques). Enfin, la méthode $setGlobalMiddleValue(1)$ permet d'initialiser une valeur à 1 au centre du maillage. Notons qu'il aurait également été possible d'initialiser ce maillage par un fichier.

La figure 4.6 donne ensuite le code de l'opération *laplacien* qui est appliquée dans la fonction main. Cette opération calcule le schéma numérique (4.9) et est appliquée à chaque itération de temps. Tout d'abord un itérateur est initialisé au début de la *DMatrix* d'entrée. Un autre itérateur est initialisé à la fin de cette même *DMatrix* (lignes 4 et 5). Pour chaque élément du maillage (ligne 6), le schéma est calculé (ligne 8 à 10) et le résultat est écrit dans la *DMatrix* de sortie (ligne 11). Les macros C++ "BEGINApplyUnary" et "END" aux lignes 1 et 13, servent à identifier le début et la fin de la définition de l'opération *laplacien*.

Cet exemple illustre que le code SkelGIS reste très proche d'un code séquentiel. Aucune difficulté n'est introduite dans ce code puisque les interfaces et les paramètres demandés sont connus de l'utilisateur. Il peut aussi être noté qu'aucune utilisation de pointeurs n'est nécessaire dans le code SkelGIS, ce qui simplifie son utilisation. La bibliothèque gère en effet elle-même la création et la destruction des pointeurs dont elle

```
1  BEGINApplyUnary(laplacien,m,double,1,m2,double,1)
2  {
3    double a = 0.05;
4    iterator<double,1> it = m.begin();
5    iterator<double,1> itEnd = m.end();
6    for(it; it<itEnd; ++it)
7      {
8        double val = m[it];
9        double res = (1-4*a)*val+a*(m.getRight(it) + m.getLeft(it)+
10                     m.getUp(it) + m.getDown(it));
11       m2[it] = res;
12     }
13 END(laplacien)
```

FIGURE 4.6 – *Opération Laplacien pour la résolution de l'équation de la chaleur.*

a besoin. Aucune notion de parallélisme n'est demandée à l'utilisateur et un style de programmation séquentiel est conservé.

4.2.3 Résultats

La résolution de l'équation de la chaleur est un problème classique et "jouet" de parallélisation de simulations scientifiques. La version parallèle SkelGIS, que nous appellerons *Heat_SK*, est très simple à mettre en œuvre, comme cela est illustré dans le code des algorithmes 4.5 et 4.6. Elle est également très proche du code séquentiel de la simulation. Cette implémentation parallèle a été comparée en terme de performances et en terme de difficulté de programmation avec une parallélisation MPI classique de l'équation de la chaleur, que nous appellerons *Heat_MPI*. Les deux implémentations sont basées sur le même code séquentiel, ce qui garantit que les deux versions sont comparables en terme de performances. En effet, la version MPI consiste en une parallélisation SPMD, ce qui implique une décomposition du domaine sur les différents processeurs (et donc une modification des structures de données) et des communications MPI entre les processeurs. En revanche, une parallélisation SPMD ne modifie pas les codes de calcul. La version SkelGIS ne modifie elle aussi que les structures de données utilisées.

	Géo-Hyd
Processeur	2×Intel Xeon Westmere (2.4 GHz)
Cœurs/nœud	8
Mémoire/nœud	24 GB
Compilateur [-O3]	OpenMPI
Réseau	Gigabit ethernet

TABLE 4.1 – *Spécifications matérielles du cluster de Géo-Hyd*

Trois expériences ont été menées sur les deux implémentations parallèles Heat_SK

	Taille du maillage	Nombre d'itérations
Expérience 1	$1k \times 1k$	$10k$
Expérience 2	$1k \times 1k$	$100k$
Expérience 3	$10k \times 10k$	$10k$

TABLE 4.2 – *Expériences de performance sur Heat_MPI et Heat_SK.*

Cœurs	Exp1 (sec)		Exp2 (sec)		Exp3 (sec)	
	MPI	SkelGIS	MPI	SkelGIS	MPI	SkelGIS
1	626.58	413.87	3760.44	3862.42	85996.4	45221.6
4	138.99	98.53	940.11	935.36	21658	10886.05
8	64.73	50	300.47	458.11	10624.625	5376.36
16	38.5	27.48	156.34	241.05	4282.22	2724.53
32	28.1	15.52	88.33	131.86	1995.78	1421.09
64	18.9	10.05	61.43	83.99	1059.13	780.99

TABLE 4.3 – *Temps d'exécution (en secondes) obtenus pour l'ensemble des expériences sur Heat_MPI et Heat_SK.*

et Heat_MPI et sont résumées dans la table 4.2. Elles consistent à faire varier la taille du domaine d'étude et le nombre d'itérations en temps et d'observer l'impact sur les performances. Ces expériences ont été effectuées sur le cluster de la société Géo-Hyd dont les caractéristiques matérielles sont indiquées dans la table 4.1. Chaque expérience a été effectuée quatre fois et moyennée. Enfin, l'écart type noté sur l'ensemble des expériences n'a pas excédé 2%. Les temps d'exécution obtenus sont répertoriés dans la table 4.3. Les valeurs rouges de ce tableau indiquent les endroits où Heat_SK obtient de moins bon temps d'exécution que Heat_MPI.

Une représentation graphique de ces résultats est également présentée dans les figures 4.7, 4.8 et 4.9. Ces figures représentent les temps d'exécution obtenus avec une échelle logarithmique. Cette représentation permet de comparer à la fois les temps d'exécution et les accélérations des deux implémentations.

Ces résultats nous permettent de faire plusieurs observations. Tout d'abord, nous pouvons noter que les résultats des deux implémentations sont très proches, ce qui montre que pour un code utilisateur séquentiel donné, une implémentation MPI ou SkelGIS donne des performances similaires. Nous pouvons également noter que pour les expériences 1 et 3 les temps d'exécution de Heat_SK sont meilleurs que les temps d'exécution de Heat_MPI. Ce résultat est dû à l'utilisation de la structure de données DMatrix que l'utilisateur n'a pas à gérer et qui est optimisée et réfléchie pour les simulations scientifiques. L'utilisation des DMatrix peut être considérée comme une optimisation qui n'est pas présente dans le code MPI, et la comparaison peut donc paraître injustifiée, cependant le code séquentiel écrit par l'utilisateur reste le même et aucun effort de programmation n'est demandé pour utiliser l'objet DMatrix. Il s'agit donc d'une optimisation implicite. La

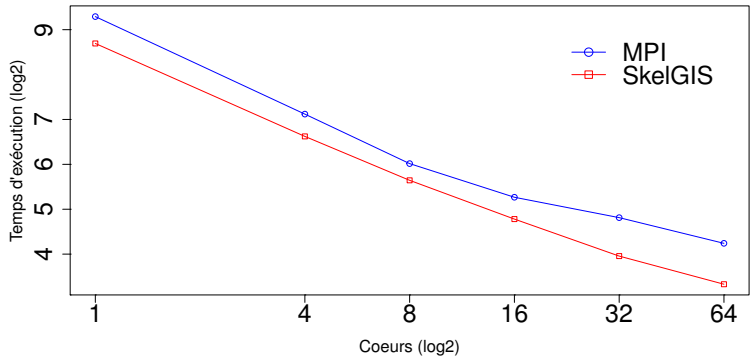


FIGURE 4.7 – *Logarithme des temps d'exécution de l'expérience 1 pour Heat_MPI et Heat_SK.*

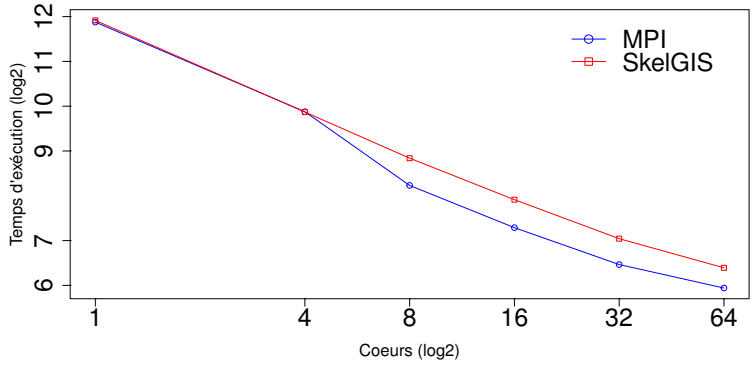


FIGURE 4.8 – *Logarithme des temps d'exécution de l'expérience 2 pour Heat_MPI et Heat_SK.*

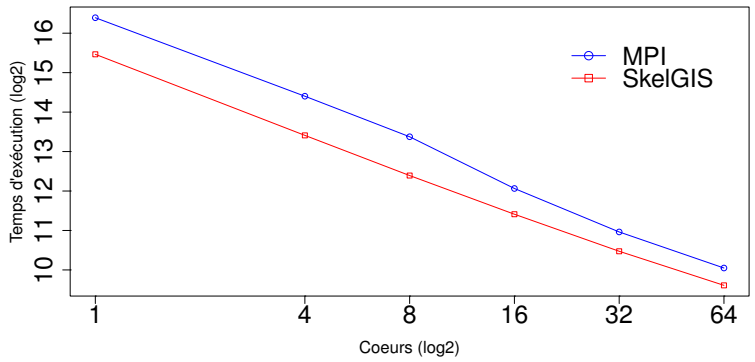


FIGURE 4.9 – *Logarithme des temps d'exécution de l'expérience 3 pour Heat_MPI et Heat_SK.*

même optimisation dans la version MPI n'aurait pas été implicite et aurait demandé du travail et des connaissances que les numériciens n'ont pas obligatoirement.

Les deux versions restent donc comparables en terme de codage. Notons, en revanche que pour l'expérience 2, les temps d'exécution de Heat_SK sont moins bons que ceux

de Heat_MPI. Nous verrons par la suite que ce problème se généralise (mais de façon moins flagrante) à des simulations plus complexes également. Ce résultat illustre que, sur un petit domaine, un grand nombre d'itérations nuit aux performances de SkelGIS. Cela est dû aux surcoûts introduits par la solution implicite, et plus spécifiquement par l'appel aux applicateurs et par la création d'itérateurs. En effet, à chaque itération de temps, un applicateur est appelé pour appliquer une fonction séquentielle utilisateur. Un applicateur va tout d'abord effectuer les communications nécessaires aux calculs puis appeler l'opération. Une opération, quant à elle, va créer et utiliser un certain nombre d'itérateurs pour parcourir le maillage. Si le maillage est petit et le nombre d'itérations grand, les surcoûts sont multipliés un grand nombre de fois, et les calculs sur les maillages, trop petits, ne permettent pas de compenser ces surcoûts. La figure 4.10 illustre l'accélération de Heat_SK pour les trois expériences. On peut y noter de façon plus évidente l'amélioration de l'accélération n augmentant la taille du maillage d'entrée.

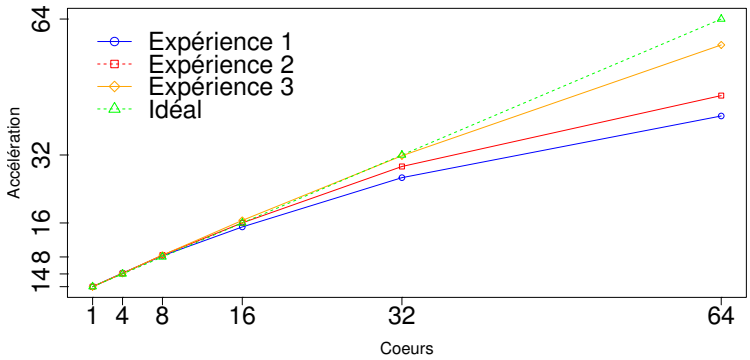


FIGURE 4.10 – *Accélération de la simulation SkelGIS pour les trois expériences.*

Les performances de la bibliothèque SkelGIS sont donc très intéressantes même sur le cas simple de l'équation de la chaleur où les surcoûts auraient pu être bien plus importants. Un deuxième type de comparaison est très important pour évaluer la qualité d'une solution de parallélisme implicite, la difficulté de programmation engendrée par la solution. En effet, le but d'une solution de parallélisme implicite est de cacher intégralement les techniques de parallélisation à l'utilisateur, tout en conservant une utilisation simple de la solution finale. En d'autres termes, si les difficultés du parallélisme sont déportées sur des difficultés d'un autre type, la solution n'a pas de réel intérêt. Pour évaluer la difficulté de codage de SkelGIS, nous avons utilisé les métriques de Halstead qui ont été présentées dans la section 2.5. Les résultats obtenus sont indiqués dans la table 4.4. Nous pouvons clairement observer que l'effort de programmation E est quarante fois moins important pour Heat_SK que pour Heat_MPI. Nous pouvons également noter que cet écart est dû aux deux facteurs V et D, qui représentent respectivement le volume de code à fournir et la difficulté du code.

L'ensemble de ces résultats montrent donc que SkelGIS atteint ses objectifs, aussi bien en terme de performance, qu'en terme d'effort et de difficulté de programmation. Une solution SIPSim pour les maillages cartésiens peut donc s'avérer très intéressante.

Métriques	MPI	SkelGIS	Gain %
N_1	513	90	82
N_2	1523	219	85.6
η_1	97	47	51.5
η_2	160	59	63
V	16.3K	2.1K	87
D	461	87	81
E	7.52M	0.18M	97.6

TABLE 4.4 – *Métriques de Halstead mesurées pour Heat_SK et Heat_MPI.*

Toutefois, la résolution de l'équation de la chaleur reste un cas très simple d'utilisation de SkelGIS où très peu d'objets SkelGIS sont nécessaires et où le calcul est très rapide. Afin de mieux apprécier les performances de la bibliothèque SkelGIS, et donc d'une solution SIPSim pour les maillages cartésiens, un cas de simulation compliqué est traité dans la dernière partie de ce chapitre.

4.3 Résolution numérique des équations de Saint Venant

Dans cette section est détaillé un cas réel d'utilisation de SkelGIS pour les maillages cartésiens à deux dimensions. La simulation présentée étudie l'écoulement de l'eau sur des surfaces planes, suivant deux dimensions. Le logiciel qui a été parallélisé se nomme FullSWOF2D [2], il est développé au laboratoire MAPMO de l'Université d'Orléans. La parallélisation de ce logiciel a été effectuée en collaboration avec Stéphane Cordier, Olivier Delestre et Minh-Hoang Le [41,43] dans le cadre du CEMRACS 2012. Dans cette partie, les informations utiles à la compréhension de l'application seront présentées, mais nous n'entrerons que peu dans les détails mathématiques de la simulation et de la résolution numérique qui dépassent le cadre de cette thèse.

4.3.1 Équations de Saint Venant

Le système d'équations de Navier-Stokes [73] permet d'étudier l'évolution temporelle d'un fluide compressible, visqueux, de densité $\rho(x, y, z, t)$ et de vitesse $w(x, y, z, t)$, dans un domaine Ω de l'espace $\mathbb{R}^3$, avec $(x, y, z) \in \Omega$ et $t \in [0, T]$. Ces équations modélisent donc les écoulements de fluides en trois dimensions. Lorsque ces équations sont intégrées suivant l'axe vertical z, c'est-à-dire lorsque une solution de la simulation est moyennée suivant l'axe z, les équations deviennent à deux dimensions et sont alors communément

appelées les équations de Saint Venant [48]. Le système conservatif qui représente l'écoulement de l'eau dans une rivière, ou un fleuve, est composé de trois EDP :

$$\begin{cases} \frac{\partial h}{\partial t} + \frac{\partial (hu)}{\partial x} + \frac{\partial (hv)}{\partial y} = 0 \\ \frac{\partial (hu)}{\partial t} + \frac{\partial}{\partial x}(hu^2 + \frac{1}{2}gh^2) + \frac{\partial (huv)}{\partial y} = -hg\frac{\partial z_b}{\partial x} \\ \frac{\partial (hv)}{\partial t} + \frac{\partial (huv)}{\partial x} + \frac{\partial}{\partial y}(hv^2 + \frac{1}{2}gh^2) = -hg\frac{\partial z_b}{\partial y} \end{cases} \tag{4.10}$$

où $h(x, y, t)$ représente la hauteur de l'eau, le vecteur (u, v) la vitesse de l'écoulement suivant les deux dimensions, g l'accélération due à la gravité, et $z_b(x, y)$ la hauteur d'eau de la rivière au repos.

4.3.2 Résolution numérique et programmation

Soit $W = (h, hu, hv)^T$ une solution des équations de Saint Venant (4.10). Le système d'équations peut s'écrire en une unique équation de la forme

$$\frac{\partial W}{\partial t} + \frac{\partial F(W)}{\partial x} + \frac{\partial G(W)}{\partial y} = S_x(W) + S_y(W) \tag{4.11}$$

où

$$F(W) = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix}, \; G(W) = \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix},$$

$$S_x(W) = \begin{pmatrix} 0 \\ -gh\frac{\partial zb}{\partial x} \\ 0 \end{pmatrix}, \; S_y(W) = \begin{pmatrix} 0 \\ 0 \\ -gh\frac{\partial zb}{\partial y} \end{pmatrix}$$

L'équation (4.11) correspond alors à l'équation (2.9) de la partie 2.2.4.2 mais sur deux dimensions. La méthode des volumes finis peut donc être utilisée pour résoudre le système d'équations.

Étant donné $W_{jk}(t)$ une solution approchée de W sur la cellule $c_{jk} = [(j - \frac{1}{2}\Delta x), (j + \frac{1}{2}\Delta x)] \times [(k - \frac{1}{2}\Delta y), (k + \frac{1}{2}\Delta y)]$ d'un maillage cartésien, et au temps t, telle que

$$W_{jk}(t) \cong \frac{1}{|c_{jk}|} \int_{c_{jk}} W(x, y, t) \mathrm{d}x\mathrm{d}y$$

où $|c_{jk}|$ représente l'aire de la cellule (en deux dimensions). On obtient le schéma numérique suivant :

$$\frac{\mathrm{d}}{\mathrm{d}t} W_{jk}(t) + \frac{1}{\Delta x}\left(F^L_{j+1/2,k} - F^R_{j-1/2,k}\right) + \frac{1}{\Delta y}\left(G^L_{j,k+1/2} - G^R_{j,k-1/2}\right) = 0 \tag{4.12}$$

où $F^{L,R}_{j\pm 1/2,k}$, $G^{L,R}_{j,k\pm 1/2}$ sont les flux numériques aux interfaces de la cellule c_{jk}, à droite (noté R) et à gauche (noté L). On retrouve alors l'équation (2.14) introduite et expliquée dans la partie 2.2.4.2 de l'état de l'art.

Dans le logiciel FullSWOF2D, la méthode des volumes finis appliquée sur un maillage cartésien permet donc de résoudre les équations de Saint Venant. Plus de détails sur cette résolution numérique sont donnés dans le livre de Bouchut [20] ainsi que dans la thèse de Delestre [48]. Notons que le schéma numérique qui vient d'être présenté est appliqué à l'ordre 2, dans la simulation, par la méthode de Heun [73]. De plus, afin d'améliorer le schéma numérique obtenu et de rendre la solution du système stable, la technique de reconstruction hydrostatique est utilisée et détaillée dans le livre de Bouchut [20]. Le pseudo-code séquentiel de cette simulation est illustré dans l'algorithme 6. Cet algorithme ressemble à l'algorithme général qui décrit une simulation scientifique, et qui a été présenté dans l'algorithme 1 de la partie 2.2.5, toutefois il est davantage détaillé pour illustrer le cas particulier de cette simulation. La première particularité est que pour être appliqué à l'ordre 2 avec la méthode de Heun, l'ensemble des schémas numériques sont appliqués deux fois, d'où la présence d'une boucle *for* supplémentaire. Le calcul du schéma numérique est ensuite un peu plus détaillé avec la séparation des calculs des conditions limites, du schéma de reconstruction hydrostatique, des flux numériques, du schéma numérique en lui-même et enfin des frottements. Nous différencions ces calculs car il s'agit pour chacun d'eux de calculs significatifs et représentant une taille importante de code.

Algorithme 6 : Algorithme séquentiel des équations de Saint Venant.

```
Création ou lecture du maillage cartésien
Création des quantités appliquées sur le maillage
Initialisation des quantités et des paramètres
Définition du pas de temps : t
Définition du temps maximal : tmax
while t < tmax do
    for i dans [[0, 2[[ do
        for chaque élément de la bordure physique do
            Calcul des conditions limites
        end
        for chaque élément du maillage do
            Calcul de la reconstruction hydrostatique
            Calcul des flux numériques
            Calcul du schéma numérique
            Calcul des frottements
        end
        Méthode de Heun pour le calcul à l'ordre 2
    end
end
```

Nous allons maintenant expliquer la parallélisation de cet algorithme en utilisant SkelGIS pour les maillages cartésiens.

4.3.3 Parallélisation avec SkelGIS

La simulation qui vient d'être décrite sur les équations de Saint Venant s'applique parfaitement à l'implémentation de SkelGIS pour les maillages cartésiens. Nous allons donc décrire comment paralléliser cette simulation en utilisant SkelGIS. Toutefois, cette simulation est complexe et représente une grande quantité de code et de détails numériques, il est donc difficile de décrire entièrement sa parallélisation avec SkelGIS comme nous l'avons fait avec l'équation de la chaleur. Nous allons donc aborder les points les plus importants de cette parallélisation.

Tout d'abord, il est important de décrire comment vont être instanciées les quantités à simuler. Dans cette simulation, pas moins de trente-sept variables appliquées au maillage sont nécessaires aux calculs. Nous ne décrirons ici que les trois quantités principales, h, u et v, et non les autres variables nécessaires au calcul. Tout d'abord, les quantités à simuler doivent être calculées en double précision. Le type de données dans les instances DMatrix est donc *double*. De plus, comme cela a été décrit dans la partie précédente, le schéma numérique appliqué dans la simulation est calculé à l'ordre 2, afin d'offrir une solution plus précise. Cette précision se traduit par un calcul stencil se servant des éléments voisins à une distance de 2 autour de la cellule courante. Enfin, le type de voisinage nécessaire à cette simulation est un voisinage en étoile, ou *star*. L'instanciation de l'objet DMatrix pour les quantités h, u et v est décrite dans la figure 4.11. Ainsi, le paramètre T est de

```
DMatrix<double,2> h
DMatrix<double,2> u
DMatrix<double,2> v
```

FIGURE 4.11 – *Déclaration des variables h, u et v*

type *double*, le paramètre Or est égal à 2 et la valeur par défaut du paramètre *box* est utilisée. Notons que les autres variables de la simulation sont, pour la plupart, soit du type $DMatrix < double, 2 >$ soit du type local $DMatrix < double, 0 >$.

L'algorithme 7 illustre la fonction principale de la simulation. Bien entendu, FullSWOF2D est un logiciel complexe écrit en langage objet C++, la fonction *main* de cette application ne ressemble donc pas réellement à celle présentée ici. Toutefois, nous essayons ici de mettre en avant les concepts de l'utilisation de SkelGIS. La fonction principale d'une telle simulation consisterait donc, tout d'abord, en l'instanciation de l'objet *DMatrix* pour les trois quantités simulées ainsi que pour l'ensemble des variables nécessaires aux calculs de la simulation. Tout comme dans le programme séquentiel, ces variables et quantités seraient ensuite initialisées. Pour cela des opérations peuvent être définies et appelées par l'intermédiaire d'applicateurs. Il est aussi possible de mettre une valeur par défaut dans ces variables, ou encore de les initialiser à l'aide d'un fichier de données. Une fois ces initialisations effectuées, la boucle principale de la simulation en temps est démarrée. Elle est suivie, tout comme dans l'algorithme séquentiel, d'une boucle *for* de deux itérations permettant d'appliquer les schémas à l'ordre 2 pour plus de précision. Dans cette boucle est ensuite appelé un applicateur. Notons ici qu'un ensemble d'applicateurs aurait aussi pu être appelé pour partitionner la simulation et l'organiser de façon plus

claire. Tout dépend du code séquentiel initial et des dépendances entre les données, tout comme dans l'implémentation séquentielle. Nous présentons ici une solution ne faisant appel qu'à un applicateur, pour simplifier l'explication. La véritable implémentation de FullSWOF2D se décompose en plusieurs appels à des applicateurs répartis dans différents objets du code. Pour terminer, un deuxième applicateur est appelé afin d'effectuer les calculs nécessaires pour appliquer le schéma à l'ordre 2.

Algorithme 7 : Fonction main codée par l'utilisateur

```
DMatrix < double, 2 > h
DMatrix < double, 2 > u
DMatrix < double, 2 > v
...
Initialisation des quantités et des paramètres
Définition du pas de temps : t
Définition du temps maximal : tmax
while t < tmax do
    for i dans [[0, 2[[ do
        apply_list({h,u,v,etc.},fullswof)
    end
    apply_list({h,u,v,etc.},heun)
end
```

Le premier applicateur appelé dans l'algorithme 7 va donc être en charge d'appliquer une opération contenant le calcul des conditions limites, de la reconstruction, des flux, du schéma numérique et des frottements. Cette opération est décrite dans l'algorithme 8.

Pour rappel, la méthode SIPSim recommande de proposer au moins deux itérateurs différents dans l'interface de programmation, le premier pour parcourir les éléments de la bordure physique sans surcoût de conditions, et le deuxième pour les autres éléments du maillage. Cette recommandation a été suivie dans l'implémentation de SkelGIS pour les maillages cartésiens, comme cela a été décrit dans la partie 4.1.3. Pour cette raison, l'opération *fullswof* va tout d'abord parcourir les éléments de la bordure physique du maillage grâce aux itérateurs adaptés (qui sont en fait au nombre de quatre). L'opérateur [] et les fonctions de voisinage sont ensuite utilisés afin de calculer les conditions limites de la simulation. Par la suite, l'itérateur contigu mis en place dans SkelGIS est utilisé afin de naviguer dans l'ensemble des éléments du maillage qui ne font pas partie de la bordure physique. Une fois encore, l'opérateur [] et les fonctions de voisinage sont utilisés afin de calculer, le schéma de reconstuction, les flux numériques, le schéma numérique et enfin les frottements appliqués à l'écoulement du fluide. Notons, de nouveau, qu'il s'agit d'une simplification de la simulation. En effet, le calcul complet de la reconstruction hydrostatique est nécessaire avant le calcul des flux numériques, par exemple. Il n'est donc en théorie pas possible de calculer les deux informations pour un même élément du maillage dans la même boucle. Comme nous l'avons évoqué précédemment, suivant le code de la simulation, plusieurs applicateurs peuvent être appelés dans la fonction principale.

Algorithme 8 : Opération séquentielle permettant de décrire le calcul des schémas numériques des équations de Saint Venant.

```
Data : {DMatrix}
Result : Modification de {DMatrix}
ItB := itérateur de début sur les bordures physiques
endItB := itérateur de fin sur les bordures physiques
while ItB≤endItB do
    Application des conditions limites avec : h[ItB], u[ItB], v[ItB],
    h.getRight(ItB,1), v.getX(ItB,2) ...
    ItB++
end
It= itérateur de début sur les éléments du maillage
endIt := itérateur de fin sur les éléments du maillage
while It≤endIt do
    Calculs avec h[It], u[It], v[It], h.getRight(It,1), v.getX(It,2) ...
    Calcul de la reconstruction hydrostatique
    Calcul des flux numériques
    Calcul du schéma numérique
    Calcul des frottements
    It++
end
```

Chaque applicateur effectuera les boucles et les calculs dont il est responsable. Ces choix d'implémentation sont à la charge de l'utilisateur tout comme s'il codait un algorithme en séquentiel.

4.3.4 Résultats

Nous allons maintenant présenter les résultats obtenus sur la simulation des équations de Saint Venant (4.10) décrites précédemment. Ces expériences sont basées sur deux implémentations parallèles du logiciel FullSWOF. Ces deux implémentations ont été effectuées de façon simultanée pendant le projet CEMRACS 2012 (Centre d'Eté Mathématique de Recherche Avancée en Calcul Scientifique) et dans une durée limitée d'environ trois semaines. La première est une version MPI, que nous appellerons FS_MPI, et la deuxième la version SkelGIS, que nous appellerons FS_SK. Ces deux implémentations sont basées sur une même version séquentielle du logiciel FullSWOF2D. FS_MPI a été implémenté par un ingénieur en mathématiques appliquées ayant les connaissances de base sur MPI, la version FS_SK, après une courte période de formation sur les concepts de SkelGIS, a été implémentée par un numéricien. FS_MPI a été implémenté de façon standard en MPI, en utilisant une décomposition de domaine, une topologie cartésienne ainsi que des types dérivés. Cette version MPI part du même code séquentiel que l'implémentation SkelGIS, et tout comme pour l'équation de la chaleur, les codes séquentiels de calculs ne sont ni modifiés ni optimisés dans ces versions parallèles. Seules les struc-

tures de données sont ré-implémentées pour être distribuées sur les processeurs. Le code séquentiel de calcul n'a donc pas été modifié ou amélioré, les deux versions parallèles du code sont donc comparables. En revanche, il ne s'agit pas des versions parallèles les plus efficaces possibles pour cette simulation. Nous ne proclamons d'ailleurs pas que SkelGIS soit aussi efficace qu'un code parallèle spécifiquement optimisé pour les équations de Saint Venant. L'objectif du modèle SIPSim, et donc de SkelGIS, est de permettre de coder en séquentiel un programme qui sera parallèle. L'efficacité du programme dépendra donc de l'efficacité du code séquentiel en lui-même.

Les expériences menées et présentées ici sont de deux types. Tout d'abord FS_MPI et FS_SK sont comparés en terme de performances sur les nœuds *thin nodes* du super-calculateur TGCC-Curie du CEA, vingtième dans le classement *top500* de novembre 2013. Son architecture matérielle est détaillée dans la table 4.5. Chaque expérience a été effectuée quatre fois et moyennée. L'écart type noté sur l'ensemble des expériences n'a pas excédé 2%. Par la suite, les métriques de Halstead [65], déjà présentées dans la partie 2.5, sont utilisées afin de comparer les deux implémentations en terme de difficulté de codage.

Calculateur	**TGCC Curie**
Processeur	2×SandyBridge (2.7 GHz)
Cœurs/nœud	16
Mémoire/nœud	64 GB
Compilateur [-O3]	Bullxmpi
Réseau	Infiniband

TABLE 4.5 – *Spécifications matérielles des nœuds* thin *du TGCC-Curie*

4.3.4.1 Performances

Nous allons tout d'abord comparer en terme de performances FS_MPI et FS_SK qui sont deux versions parallèles comparables car basées sur le même code séquentiel. Quatre expériences ont été menées pour évaluer les performances de ces simulations parallèles et sont décrites dans la table 4.6. Ces expériences font varier la taille du domaine (expériences 1, 3 et 4) ou le nombre d'itérations en temps (expériences 1 et 2). L'ensemble des temps d'exécution obtenus (en secondes) sont présentés dans la table 4.7. Une représentation graphique de ces résultats est également proposée dans les figures 4.12, 4.13, 4.14 et 4.15.

Pour l'ensemble des expériences effectuées nous pouvons observer des similarités. Tout d'abord, nous pouvons noter que les temps d'exécution obtenus pour FS_SK sont, hormis les valeurs en rouge, meilleurs que pour FS_MPI. Etant donné que la même version séquentielle de code a été utilisée au départ, il s'agit d'une remarque intéressante pour la solution SkelGIS. En effet, cette performance provient de l'objet DMatrix. Cet objet est le seul objet qui n'est pas utilisé dans la version séquentielle, et qui peut être producteur

	Taille du maillage	Nombre d'itérations
Expérience 1	$5k \times 5k$	$5k$
Expérience 2	$5k \times 5k$	$20k$
Expérience 3	$10k \times 10k$	$5k$
Expérience 4	$20k \times 20k$	$5k$

TABLE 4.6 – *Expériences de performance sur FS_MPI et FS_SK.*

Cœurs	Exp1 (sec)		Exp2 (sec)		Exp3 (sec)		Exp4 (sec)	
	MPI	SkelGIS	MPI	SkelGIS	MPI	SkelGIS	MPI	SkelGIS
32	4868.43	3494.92						
64	2317.7	1780.4	9353.47	7391.96				
128	1154.25	898.219	4578.28	3768.54	45588	31332.9		
256	578.65	460.715	2282.53	1988.8	22089.9	17385.6	90974,2	68017.1
512	277.39	284.585	1118.01	1117.09	11299.4	9436.11	45487.1	35112.2
1024	144.26	155.85	557.621	602.66	5739.52	5127.93	22299.1	19821.3
2048	67.49	103.363	273.785	407.73	2930.48	3196.71	11867.4	10986.8

TABLE 4.7 – *Temps d'exécution (en secondes) obtenus pour l'ensemble des expériences sur FS_MPI et FS_SK.*

d'une amélioration de performances (et non de surcoûts) de la version FS_SK. Dans la version FS_SK, l'utilisateur n'a plus à se soucier de coder ses propres structures de données et leur accès efficace. Ce travail est confié à la bibliothèque SkelGIS, ce qui simplifie tout d'abord le code, et ce qui permet d'obtenir une structure optimisée "gratuitement", sans aucun effort. Ce constat a d'ailleurs également été fait pour l'équation de la chaleur.

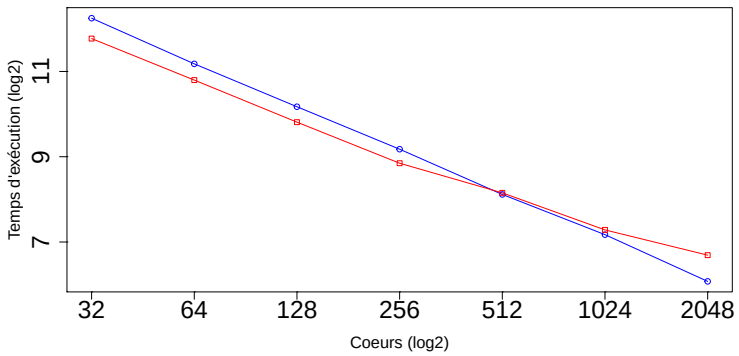


FIGURE 4.12 – *Logarithme des temps d'exécution de l'expérience 1 pour FS_MPI et FS_SK.*

Cependant, nous pouvons également noter que la pente des courbes des figures 4.12, 4.13, 4.14 et 4.15 est plus accentuée et donc meilleure pour FS_MPI que

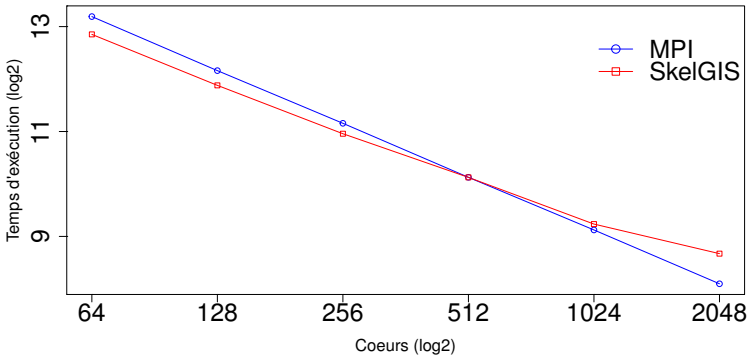


FIGURE 4.13 – *Logarithme des temps d'exécution de l'expérience 2 pour FS_MPI et FS_SK.*

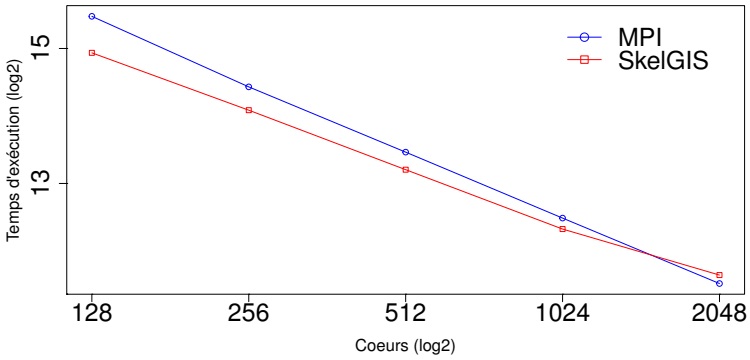


FIGURE 4.14 – *Logarithme des temps d'exécution de l'expérience 3 pour FS_MPI et FS_SK.*

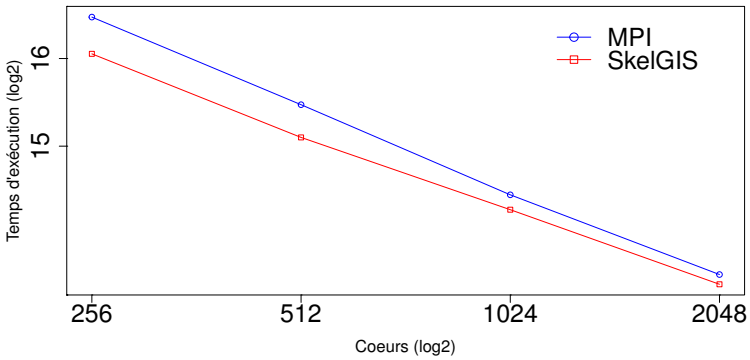


FIGURE 4.15 – *Logarithme des temps d'exécution de l'expérience 4 pour FS_MPI et FS_SK.*

pour FS_SK. En effet, ces figures représentent les temps d'exécution sur une échelle logarithmique. Cette représentation a la particularité de permettre de comparer les temps d'exécution mais également de connaître la linéarité du speedup de la simulation. Si le temps d'exécution présenté est linéaire, le speedup le sera également, et la pente du

temps d'exécution représente également la pente du speedup. Ainsi, il semble que pour l'ensemble des expériences, le speedup de FS_MPI soit légèrement meilleur que celui de FS_SK. Ce phénomène traduit, à l'inverse, les surcoûts de la solution SkelGIS, comme cela a été noté pour l'équation de la chaleur également. En effet, si l'objet DMatrix peut, de son côté, être à l'origine d'un gain de performances, du simple fait que la structure de données qui y est implémentée est plus efficace, les autres objets SkelGIS tels que les applicateurs et les itérateurs provoquent des appels supplémentaires et donc des surcoûts. Notons que ces surcoûts sont plus accentués sur les expériences 1 et 2, ce qui montre qu'il y a plus d'impact sur le nombre d'itérations en temps que sur la taille du domaine à traiter. Ce phénomène s'explique assez bien. À chaque itération de temps, des applicateurs sont appelés. Comme cela a déjà été expliqué, les applicateurs procèdent, tout d'abord, aux communications nécessaires pour les calculs, puis, appellent l'opération de l'utilisateur. Cette opération va ensuite créer des itérateurs et parcourir le maillage. Si le nombre d'itérations en temps augmente, les surcoûts liés aux applicateurs et aux itérateurs sont multipliés, alors que si la taille du domaine augmente, seul le parcours des éléments du maillage est plus long mais les surcoûts, eux, restent identiques.

Pour conclure sur les performances de SkelGIS, il semble tout d'abord évident que, partant d'un code séquentiel commun, SkelGIS obtient de très bonnes performances aussi bien en temps d'exécution qu'en passage à l'échelle. SkelGIS offre des perspectives très intéressantes par le biais de l'objet DMatrix. De façon plus générale, la méthode SIPSim offre des performances intéressantes grâce à son composant DDS. En effet, dans le cas d'un maillage cartésien, la structure de données mise en place reste très simple. Dans le cas de maillages plus complexes, ce composant peut se montrer primordial comme cela sera illustré dans le chapitre 5. Si SkelGIS provoque malgré tout des surcoûts qui peuvent nuire à la linéarité de ses speedup, SkelGIS reste tout de même une solution de parallélisme implicite très efficace sur les maillages cartésiens et qui propose des performances proches d'une version MPI comparable.

4.3.4.2 Effort de programmation

L'approche SIPSim vise à proposer des solutions de parallélisme implicite pour les simulations scientifiques. Pour cette raison, SkelGIS se doit d'être une solution simple d'utilisation. Nous allons donc présenter les résultats qui ont été obtenus en terme de difficulté de codage, toujours en comparant FS_MPI et FS_SK. Pour ce faire, les métriques de Halstead [65] ont de nouveau été mesurées pour ces deux versions parallèles de FullSWOF. Les résultats obtenus sont présentés dans la table 4.8.

Nous pouvons tout d'abord observer, dans ces résultats, que l'effort de programmation E à fournir est environ vingt fois moins important pour FS_SK que pour FS_MPI. Ce résultat montre donc que l'ambition de SIPSim, pour proposer des solutions de parallélisme implicite simples, est atteinte. Rappelons que le résultat de l'effort de programmation dans les métriques de Halstead est égal à la multiplication du volume du programme V par la difficulté de codage du programme D. Nous pouvons observer dans ce résultat que le volume de code V produit dans FS_SK est environ cinq fois moindre que dans FS_MPI et que la difficulté D est environ quatre fois moindre dans FS_SK

Métriques	MPI	SkelGIS	Gain %
N_1	7895	2673	66
N_2	45147	8507	81
η_1	414	297	28.3
η_2	414	353	14.7
V	537.1K	104.5K	80.5
D	13274	3576	73
E	7130M	373M	94.7

TABLE 4.8 – *Métriques de Halstead mesurées pour FS_MPI et FS_SK.*

que dans FS_MPI. Ce résultat est intéressant puisqu'il montre deux aspects différents de simplicité d'utilisation dans SkelGIS. Tout d'abord, le volume du programme parallèle de FullSWOF écrit avec SkelGIS est cinq fois moins important que le volume du programme MPI. Nous avons d'ores et déjà expliqué ce résultat, il est dû à l'utilisation de l'objet DMatrix. En effet, l'utilisation de cet objet permet de s'abstraire de la programmation d'une structure de données dans le code utilisateur, cette structure de données étant entièrement gérée par SkelGIS. De plus, la répartition de cette structure de données sur les différents processeurs est elle aussi entièrement implicite ce qui allège encore davantage le volume du code final. Le second point de simplicité de SkelGIS, illustré par ces résultats, est qu'il est quatre fois plus difficile de coder FS_MPI que FS_SK. Cette métrique est fortement liée au nombre total et distinct d'opérateurs et d'opérandes dans les deux implémentations. Une solution SIPSim ne fait appel qu'à quatre composants principaux et délègue dans ces quatre composants les structures de données, les décompositions de domaine et les communications MPI. Pour cette raison la complexité du code en terme de nombre de variables et d'appels de fonctions est moins importante dans FS_SK.

Pour finir, et comme cela a déjà été abordé dans l'état de l'art, les métriques de Halstead ne s'intéressent qu'à des concepts de programmation séquentiels. Ainsi, l'appel à une fonction MPI aura le même coût que l'appel à une fonction classique d'un code séquentiel. Aucune métrique existante ne tient compte de la difficulté des concepts parallèles et de la difficulté de penser le programme en parallèle. Il est très difficile de mettre en œuvre de telles métriques, toutefois il semble évident, dans ce cas, que la véritable difficulté de programmation D de FS_MPI soit supérieure à celle trouvée par les métriques de Halstead. SkelGIS de son côté, ne fait appel à aucune notion de parallélisme et est aussi simple à mettre en œuvre, conceptuellement, qu'un code séquentiel.

4.4 Conclusion

Dans ce chapitre nous avons décrit l'implémentation de SkelGIS dans le cas particulier des maillages cartésiens à deux dimensions. Nous avons tout d'abord détaillé l'implémentation de cette solution en suivant les composants du modèle SIPSim, puis nous avons

exposé deux cas d'application réels. Le premier sur la résolution de l'équation de la chaleur, et le deuxième sur la résolution des équations de Saint Venant, en suivant la méthode appliquée dans le logiciel FullSWOF2D. Un ensemble de résultats a été présenté, tout d'abord sur les performances de la solution, mais aussi sur l'effort de programmation à fournir pour utiliser SkelGIS. Ces résultats ont été comparés à ceux d'une implémentation MPI implémentée à partir d'un même code séquentiel. Le modèle SIPSim, et donc la bibliothèque SkelGIS, laissant à la charge de l'utilisateur le code séquentiel qui décrit les schémas numériques à calculer, la performance du code final dépend également du code séquentiel implémenté. Il était donc important de partir pour ces deux implémenetations d'une même version séquentielle sans y ajouter d'optimisations séquentielles particulières mais en proposant toutefois une implémentation MPI efficace. Les résultats obtenus sont très intéressants et montrent qu'il est possible de proposer une implémentation du modèle SIPSim efficace, d'autant plus que toutes les implémentations possibles n'ont pas été mises en place, comme par exemple l'utilisation des registres de vectorisation. Cette première implémentation de SkelGIS a donc illustré la viabilité du modèle SIPSim sur des maillages cartésiens à deux dimensions.

Il est donc possible, grâce au modèle SIPSim d'implémenter des solutions de parallélisme implicite efficace pour les maillages de type cartésiens. Une extension de ce travail peut d'ailleurs proposer une solution pour des maillages cartésiens à n dimensions. De plus, grâce aux travaux OP2 [59, 60, 94] et Liszt [50], nous savons que les concepts du modèle SIPSim peuvent s'appliquer au cas des maillages non-structurés. En effet, OP2 et Liszt proposent des modèles de programmation parallèle implicite proches de ceux proposés par l'approche SIPSim. Le reste de cette thèse propose une implémentation du modèle SIPSim pour un cas d'application qui n'a, à notre connaissance, jamais été traité dans des solutions de parallélisme implicite : les simulations sur des réseaux.

SkelGIS pour des simulations sur réseaux

5

Sommaire

Le chapitre précédent a présenté l'implémentation de SkelGIS pour les simulations sur des maillages cartésiens à deux dimensions. Dans ce chapitre est abordé un cas plus complexe d'implémentation de la méthode SIPSim. La bibliothèque SkelGIS a, en effet, été implémentée dans le cas de simulations sur des *réseaux* pouvant être représentés sous forme de graphes dirigés acycliques (DAG). Un réseau n'est pas considéré comme un maillage, même s'il peut s'y apparenter par certains côtés. Dans ce chapitre sera tout d'abord décrite la notion de réseau afin de comprendre pourquoi la méthode SIPSim peut être appliquée à ce type de simulations. Par la suite, l'implémentation de SkelGIS pour les réseaux sera détaillée. Cette implémentation, plus complexe que le cas des maillages cartésiens à deux dimensions, nécessitera une partie supplémentaire décrivant avec précision l'implémentation de la structure de données distribuée. Un cas d'application réel sera ensuite présenté, afin d'évaluer les performances de SkelGIS. Il s'agit d'une simulation d'écoulement du sang dans le réseau artériel. L'implémentation de cette simulation avec SkelGIS sera comparée avec une version OpenMP, et sera évaluée sur différents types de clusters. Enfin, des travaux plus récents seront présentés sur le problème de partitionnement des réseaux. Nous étudierons l'implémentation actuelle de partitionnement dans SkelGIS, puis deux méthodes plus proches du véritable problème de partitionnement des réseaux.

5.1 Les réseaux

Afin de faciliter la compréhension de ce chapitre nous allons définir ce qui est appelé un réseau et les caractéristiques qui en découlent. Cette notion est utilisée dans diverses simulations, toutefois, malgré son utilisation fréquente, les détails sur ce qu'est exactement appelé un réseau sont peu abordés dans la littérature. Nous allons illustrer ici qu'un réseau peut être assimilé, par certains côtés, à un maillage. Toutefois un réseau n'est pas un maillage pour plusieurs raisons qui seront présentées.

Tout d'abord, il est évident qu'un réseau, en terme de simulation, est une structure permettant de simuler des phénomènes réels de réseaux tels que les réseaux routiers, sanguins, fluviaux, pétrolifères etc. Un réseau permet de discrétiser le domaine en espace en deux types d'éléments différents : les nœuds et les arêtes. Dans le cas d'une simulation d'écoulement du sang dans le réseau artériel, par exemple, les arêtes du graphe sont assimilées aux artères, et les nœuds du graphe aux points de rencontre de plusieurs artères, aussi appelés conjonctions. Les points de rencontre des artères n'ayant pas tous la même connectivité, un réseau est une structure irrégulière. Cette discrétisation peut être apparentée à un graphe dirigé ou non, avec ou sans cycles et projeté dans $\mathbb{R}^2$ ou $\mathbb{R}^3$. Les deux types d'éléments sont donc les nœuds et les arêtes et à chacun de ces deux types pourront être appliqués deux discrétisations et deux schémas numériques différents (figure 5.1). Un réseau permet donc de résoudre des problèmes représentant deux phénomènes physiques différents mais liés entre eux. La notion de réseau rappelle alors les maillages avancés par blocs ou hybrides, qui ont été évoqués dans la section 2.2.2.2 de cette thèse.

Un réseau peut être assimilé à un maillage pour deux raisons :

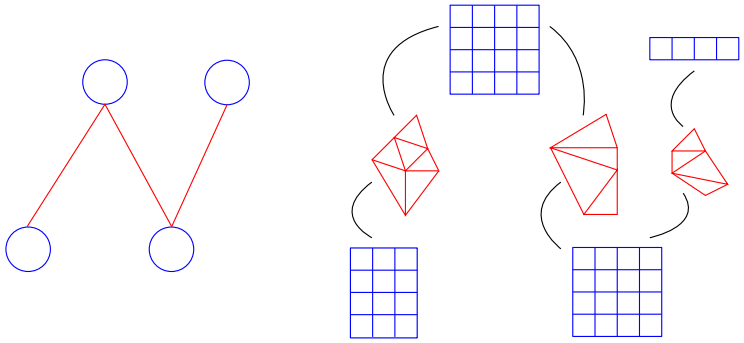


FIGURE 5.1 – *Illustration d'un réseau à gauche et d'un exemple de simulation multi-physique à droite avec deux types de discrétisation. Les nœuds subissent une discrétisation cartésienne de l'espace et les arêtes subissent une discrétisation non-structurée de l'espace.*

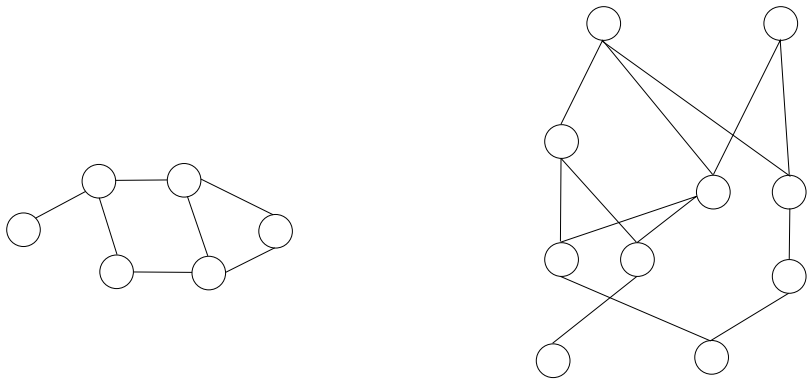
— La construction d'un réseau consiste en la dicrétisation du domaine en deux types d'éléments.
— Un réseau représente une connectivité entre ses éléments sous la forme d'un graphe.

Pour ces deux raisons, certaines ressemblances avec un maillage, en particulier avec un maillage irrégulier, peuvent être trouvées et utilisées pour l'implémentation. En revanche, un réseau n'est pas un maillage pour deux autres raisons :

— Le graphe formé par un réseau ne représente pas des faces ou des cellules, et les calculs appliqués ne porteront pas sur des cellules, contrairement aux maillages.
— Le graphe ne forme pas l'objet simulé mais le représente avec deux types d'éléments différents, potentiellement très grands, sur lesquels seront effectués des calculs qui peuvent à nouveau discrétiser l'espace.

Afin de rendre plus claire la différence entre un maillage et un réseau nous allons essayer d'en donner des définitions par l'intermédiaire de la théorie des graphes. Nous avons d'ores et déjà vu qu'un maillage est un graphe connexe, sans isthme dont la planarité est systématique dans $\mathbb{R}^2$ mais pas dans $\mathbb{R}^3$. Un réseau est un graphe connexe mais qui peut contenir des isthmes. Autrement dit, les nœuds d'un réseau ne forment pas des faces, ou cycles élémentaires. De plus un réseau, même s'il est plaqué dans $\mathbb{R}^2$, n'est pas nécessairement planaire contrairement à un maillage. Prenons deux exemples simples qui montrent qu'un réseau n'est pas obligatoirement un graphe planaire, même sur $\mathbb{R}^2$. Dans un réseau représentant les fleuves et rivières, par exemple, il arrive qu'il existe des rivières souterraines avec un point d'entrée et un point de sortie en surface, comme c'est le cas pour la résurgence de la Loire, appelée le Loiret, par exemple. Par conséquent les arêtes, qui représentent les rivières et fleuves peuvent se croiser sans pour autant qu'un point de conjonction ne soit présent à cette intersection. La même remarque peut être faite sur un réseau routier dans lequel il y aurait des tunnels souterrains, ce qui superposerait plusieurs routes qui ne se rencontrent pas. Un réseau est donc un graphe connexe quelconque. La figure 5.2(a) illustre un graphe planaire qui ne forme pas un

maillage mais un réseau, et la figure 5.2(b) illustre un réseau où des arêtes peuvent se croiser sans l'existence d'un nœud à la conjonction de ces arêtes.



(a) Graphe planaire qui ne représente pas un maillage.

(b) Graphe connexe quelconque représentant un réseau.

FIGURE 5.2 – *Maillages et réseaux.*

Dans une simulation sur les réseaux deux discrétisations du domaine peuvent être effectuées. La première pour construire le réseau et la deuxième sur chaque élément du réseau, c'est-à-dire l'ensemble des nœuds et des arêtes. De cette façon il est possible d'appliquer des schémas numériques à l'ensemble du domaine tout en définissant deux comportements différents lors de la simulation. Dans un réseau deux types de schémas numériques différents peuvent donc être appliqués, un à chaque type d'élément, les nœuds et les arêtes. Ces deux schémas numériques ont la particularité d'être liés dans une même itération de temps, c'est-à-dire que l'un des deux impacte le résultat de l'autre. Par conséquent, dans une simulation sur les réseaux, même si chaque schéma numérique est explicite et ne crée pas de dépendance parmi les éléments d'une même itération de temps (comme c'est le cas dans l'équation (2.3)), une dépendance peut être créée entre les deux schémas numériques. Il en résulte, soit l'obtention de nouveaux schémas numériques explicites, soit plus généralement l'obtention de nouveaux schémas numériques implicites (équation (2.4)). Si un ou plusieurs schémas implicites sont obtenus, suite à la mise en réseau de chaque schéma explicite, l'un des deux types d'éléments du réseau devra être calculé avant l'autre et son résultat impactera le deuxième dans une même itération de temps. Ainsi si l'on note $T1$ et $T2$ les arêtes et les nœuds du réseau, ou vice-versa, une simulation sur un réseau sera typiquement constituée de quatre étapes :

1. communication des éléments $T1$ à $t-1$,
2. calcul des éléments $T2$ à t,
3. communication des éléments $T2$ à $t-1$ ou/et t,
4. calcul des éléments $T1$ à t.

L'étape 3 est l'étape qui caractérise le fait que les schémas numériques sont explicites ou implicites. En effet, si l'étape 4, pour être calculée, a besoin des éléments $T2$ uniquement à l'instant $t-1$, nous considérons les schémas comme explicites. Si, en revanche, l'étape 4 nécessite les éléments $T2$ à l'itération t alors les schémas sont implicites. Notons que si les schémas numériques sont explicites alors les étapes peuvent être organisées différemment :

1. communication des éléments $T1$ et $T2$ à $t-1$,
2. calcul des éléments $T1$ et $T2$ à t,

La nature des schémas numériques (explicite ou implicite) n'est pas le seul élément impacté par la liaison des schémas numériques d'un réseau. En effet, la notion de voisinage $N(x)$ (ou stencil) est elle aussi modifiée. Dans les schémas numériques d'un réseau, chaque schéma numérique (implicite ou non) possède un voisinage défini par $N(x) = N_{in}(x) \cup N_{out}(x)$. $N_{in}(x)$ représente le $N(x)$ que l'on peut trouver dans une simulation classique définissant un seul maillage et un seul schéma numérique, il s'agit des éléments voisins de x dans le maillage local. $N_{out}(x)$ représente le voisinage issu du réseau. En effet, dans un réseau, comme illustré dans la figure 5.1, les différents maillages, et donc les différents schémas numériques sont liés entre eux. Cette liaison se traduit par $N_{out}(x)$. Notons qu'un élément interne au maillage local n'est pas concerné par la liaison avec un autre maillage et alors $N_{out}(x) = \emptyset$. La définition du voisinage N_{out} d'un réseau sera étudiée en détails dans la partie 5.2.4.2, et est illustrée dans la figure 5.3.

Enfin une dernière caractéristique des réseaux doit être abordée dans cette partie. Un maillage est concerné par ce qui est appelé une bordure physique. Comme il l'a été expliqué précédemment, lorsqu'une simulation est effectuée, l'espace doit être borné alors que le phénomène réel ne l'est pas. Pour cette raison des conditions limites sont ajoutées aux EDP et permettent de simuler de façon plus réaliste ce qui se passe aux bords du domaine. Un réseau, tout comme un maillage, discrétise le domaine initial, il existe donc également une notion de bordure physique dans un réseau. Dans un réseau général, représenté par un graphe quelconque, dirigé ou non, les bordures physiques vont être représentées par certains nœuds indiqués au préalable comme bordures physiques du domaine.

Dans le reste de ce chapitre, la définition générale d'un réseau n'est pas utilisée. En effet, nous nous intéressons ici à une sous-partie des réseaux qui peuvent être représentés par un *graphe dirigé acyclique*, ou DAG. Dans ce cas, les nœuds représentant la bordure physique du domaine sont les nœuds racines et les nœuds feuilles du DAG. Le reste de ce chapitre s'intéresse également à des simulations où les schémas numériques appliqués aux éléments du réseau sont définis dans une unique dimension. Ce travail donne donc une première solution de parallélisme implicite pour les simulations sur les réseaux et étudie sa faisabilité. A notre connaissance, aucun travail similaire n'existe à l'heure actuelle.

5.2 SIPSim pour les réseaux

Tout comme pour l'implémentation de SkelGIS pour les maillages cartésiens, nous allons tout d'abord décrire l'implémentation des quatre composants de la méthode SIPSim pour le cas des simulations sur les réseaux.

5.2.1 Structure de données distribuée

Tout comme pour l'implémentation de la méthode SIPSim pour les grilles cartésiennes à deux dimensions, le premier composant de la méthode SIPSim à implémenter est la structure de données distribuée (DDS) permettant de représenter le maillage et sa connectivité. Comme il l'a été décrit précédemment, un réseau ne peut pas être considéré comme un maillage, toutefois il s'en approche puisque, tout comme un maillage, il consiste à discrétiser le domaine en éléments et à en représenter leur connectivité. Pour cette raison la structure de données distribuée qui représente le réseau doit posséder les mêmes caractéristiques qu'un maillage, et la méthode SIPSim s'applique donc parfaitement à ce type de simulations. La structure de données doit donc garantir un accès efficace aux éléments (nœuds et arêtes) et aux voisinages N_{out}. Ce DDS, là encore, va être responsable de l'efficacité de la solution, et l'ensemble de l'implémentation de la méthode SIPSim pour les réseaux repose sur ce premier composant. Rappelons que nous nous intéressons ici à la sous classe des réseaux pouvant être représentés sous forme de DAG. La DDS pour ces réseaux est nommée *DDAG*. L'implémentation de ce DDS est complexe, et la section 5.3 décrira en détails cette implémentation. Bien évidemment, la difficulté majeure se trouve dans le fait qu'un réseau est une structure irrégulière où la connectivité est différente pour chaque élément, là où elle est régulière pour les grilles cartésiennes.

Une structure de données irrégulière va donc tout d'abord poser un problème d'efficacité pour accéder aux éléments et à leur voisinage. La résolution de ce premier problème sera entièrement décrite dans la section 5.3. Mais outre cette difficulté, une structure irrégulière pose un autre problème, qui lui aussi impacte de façon significative l'efficacité de la solution. En effet la méthode SIPSim implémente des solutions SPMD ce qui implique une bonne décomposition de domaine pour obtenir de très bonnes performances. Dans le cas des réseaux, cette décomposition de domaine va se traduire par un problème de partitionnement de graphe. Le problème du partitionnement de graphe est un problème connu et NP-complet, ce qui signifie que des heuristiques sont appliquées pour approcher au mieux la solution idéale en un temps raisonnable. Le partitionnement de réseau est particulier par rapport à un partitionnement de graphe classique où il est considéré que des calculs sont faits soit sur les nœuds, soit sur les arêtes. Dans le cas d'un réseau des calculs sont effectués sur les deux types d'éléments et impactent l'équilibrage de charge. L'heuristique implémentée dans le prototype actuel de SkelGIS ainsi que deux autres algorithmes seront étudiés dans la section 5.5. En effet, dans la version actuelle de SkelGIS, une heuristique de rattachement des arêtes sœurs a été implémentée. Cette solution propose des résultats raisonnablement bons, comme nous le verrons dans la partie d'évaluation 5.4, toutefois il est probable qu'avec les deux autres solutions étudiées dans

la section 5.5, qui utilisent du partitionnement d'hypergraphe, les performances soient meilleures.

Lors d'un partitionnement de graphe, et donc d'un partitionnement de réseau, la solution idéale n'est que rarement trouvée et il persiste toujours un léger déséquilibre dans la solution de partitionnement. Pour cette raison, un point essentiel pour améliorer les performances de la solution est d'opérer un recouvrement des communications avec les calculs locaux. En effet, dans un programme parallèle de type SPMD pour du calcul stencil, chaque sous-domaine en espace appartenant à chaque processeur peut être décomposé en deux parties. Une première partie est dite *locale-interne*. Cette partie peut être calculée sans aucun échange avec les autres processeurs. La deuxième partie *locale-externe*, en revanche, nécessite des communications avec les autres processeurs afin de connaître les valeurs au bord du domaine local. Dans un mécanisme de recouvrement des communications par les calculs, le programme se découpe alors en quatre étapes :

1. les communications non bloquantes MPI sont initialisées
2. les calculs sur la partie *locale-interne* sont effectués
3. les communications non bloquantes MPI sont terminées
4. les calculs sur la partie *locale-externe* sont effectués

Ce mécanisme a été mis en place pour l'implémentation de la méthode SIPSim sur les réseaux (SkelGIS pour les réseaux). Ce mécanisme est possible par l'optimisation de la structure de données distribuée qui sera décrite dans la section 5.3.

5.2.2 Application de données

L'objet *DDAG* renferme donc une structure de données lourde et complexe, contrairement à l'implémentation de l'objet *DMatrix*. Cette complexité est due au fait qu'il est nécessaire, pour chaque élément de la structure, de retenir la connectivité qui lui est propre. Chaque élément a, en effet, potentiellement une connectivité différente. Pour cette raison, l'implémentation de SkelGIS pour les réseaux reste fidèle à la méthode SIPSim. En effet, pour l'implémentation des *DMatrix* le choix a été fait d'embarquer le composant DPMap dans le DDS lui-même, celui-ci étant léger. Dans le cas des réseaux, le DDS DDAG étant beaucoup plus lourd, il sera instancié une seule fois pour chaque type de réseau de la simulation, puis des objets plus légers se chargeront d'appliquer les données sur ce DDS. Ces objets sont de deux types, un pour appliquer des données sur les nœuds du réseau, et l'autre pour appliquer des données sur les arêtes du réseau. Ils sont nommés respectivement *DPMap_Nodes* et *DPMap_Edges*. Ce sont ces objets qui seront instanciés pour définir les quantités simulées dans les schémas numériques et ainsi pour obtenir σ.

5.2.3 Applicateurs et opérations

Une fois le DDAG défini et les quantités à simuler instanciées, il faut appliquer les schémas numériques séquentiels. L'utilisateur va alors, comme pour les *DMatrix*, définir des opérations séquentielles qui vont représenter ses schémas numériques. Libre à lui de

définir autant d'opérations que nécessaire, sa seule contrainte est d'appliquer ses opérations par le biais d'applicateurs. Il en existe deux à l'heure actuelle pour les réseaux, ils s'appellent apply_list et apply_listi. Ce sont des procédures définies par :

$$apply\_list : \{DPMap\_Edges\} \times \{DPMap\_Nodes\} \times Op \tag{5.1}$$

$$apply\_listi : \{T1\} \times \{T2\} \times Op1 \times Op2 \tag{5.2}$$

où $\{DPMap\_Edges\}$, $\{DPMap\_Nodes\}$, $\{T1\}$ et $\{T2\}$ sont des ensembles d'instances des objets $DPMap\_Edges$ et $DPMap\_Nodes$, et Op, $Op1$ et $Op2$ des opérations.

Le premier applicateur effectue tout d'abord les communications afin d'obtenir les valeurs voisines, et non locales, nécessaires aux calculs des quantités (DPMap) indiquées en argument. L'applicateur appelle ensuite l'opération Op de l'utilisateur. Ce premier applicateur peut être particulièrement intéressant pour l'application de schémas numériques explicites car il a la particularité d'offrir beaucoup de libertés à l'utilisateur. En effet, l'utilisateur peut librement écrire une opération qui agira sur les nœuds comme sur les arêtes, dans l'ordre souhaité. Les schémas numériques étant explicites, seules les communications effectuées avant l'appel à l'opération sont nécessaires. Toutefois, ces opérations, très générales, ne peuvent être mises en place pour des schémas numériques implicites, puisqu'alors des communications sont nécessaires entre les phases de calcul. Cependant, il est tout de même possible d'appliquer des schémas numériques implicites avec ce premier applicateur. Dans ce cas, il sera nécessaire d'appeler l'applicateur à chaque phase du calcul afin d'effectuer implicitement les échanges de données nécessaires. Il n'est alors pas évident de définir l'utilisation de l'applicateur, ni de comprendre son utilisation. Pour cette raison, un deuxième applicateur est proposé et permet d'effectuer des calculs implicites en quatre étapes, comme présenté dans la partie 5.1. Cet applicateur commencera par (1) effectuer une première phase de communication des éléments de type $T1$ (calculés à l'itération $t-1$), puis (2) appellera l'opération $Op2$ sur les éléments de type $T2$. Par la suite, (3) une deuxième phase de communication aura lieu pour échanger les éléments de type $T2$ venant d'être calculés (à l'itération t donc), pour enfin (4) appeler l'opération $Op1$ qui effectue les calculs sur les éléments $T1$. Notons, de nouveau, que $T1$ et $T2$ peuvent alors être associés indifféremment aux nœuds ou aux arêtes du réseau.

5.2.4 Interfaces de programmation

Enfin afin de pouvoir programmer une opération le dernier composant de la méthode SIPSim doit être implémenté. Il s'agit des interfaces de programmation qui sont regroupées en trois types précédemment décrits, les *itérateurs*, les *accesseurs*, et les fonctions pour accéder aux *voisinages* des éléments.

5.2.4.1 Itérateurs

Les *itérateurs* permettent de se déplacer dans des données appliquées à un *DDS*. Dans le cas des réseaux, les itérateurs permettent donc de se déplacer dans des instances des objets *DPMap_ Nodes* et *DPMap_ Edges*. Il existe huit itérateurs pour les réseaux, trois pour l'objet *DPMap_ Edges* et cinq pour l'objet *DPMap_ Nodes*.

Rappelons que, tout comme pour les *DMatrix*, l'ordre de parcours de l'ensemble des nœuds ou de l'ensemble des arêtes du réseau n'a pas d'importance, et que la solution obtenue sera la même si l'ordre de parcours est modifié. En effet, comme expliqué dans la partie 5.1, le seul ordre de traitement peut venir des phases de calcul sur les nœuds ou sur les arêtes. Ceci est dû au fait que les deux schémas numériques explicites (2.3) de départ n'introduisent pas de dépendances entre les nœuds d'une même itération de temps, ou entre les arêtes d'une même itération de temps.

Nous allons décrire les trois itérateurs communs aux objets *DPMap_ Nodes* et *DPMap_ Edges*. Tout d'abord, un premier itérateur permet simplement de parcourir l'ensemble des nœuds/arêtes du réseau et cet itérateur garantit que l'ensemble des nœuds/arêtes est bien parcouru. Cet itérateur est ensuite divisé en deux itérateurs qui vont permettre de parcourir les nœuds/arêtes *locaux-internes* et *locaux-externes* précédemment décrits. Ces deux itérateurs seront utilisés pour effectuer le recouvrement des communications MPI par les calculs locaux dans les applicateurs. Deux itérateurs supplémentaires ont été implémentés afin de naviguer dans la bordure physique du domaine. Il a été décrit dans la section 5.1 que les bordures physiques, dans le cas d'un réseau de type DAG, étaient représentées par deux types particuliers de nœuds, les racines et les feuilles du DAG. Un premier itérateur sur les bordures physiques permet donc de naviguer dans les racines du DAG, et un deuxième dans les feuilles du DAG. Tout comme pour les *DMatrix*, ces deux itérateurs sont très importants pour éviter des conditions inutiles dans le code utilisateur afin de déterminer si le nœud courant est une racine, une feuille ou un nœud interne. Avec l'ensemble de ces itérateurs, l'utilisateur peut naviguer dans la bordure physique et dans les nœuds internes sans conditions dans le code. Notons que l'implémentation de ces itérateurs est possible, là encore, grâce aux optimisations de la DDS DDAG qui sera décrite dans la section 5.3.

5.2.4.2 Voisinages

La notion de voisinage est l'une des notions les plus importantes de la méthode SIPSim puisqu'elle rend possible les calculs *stencil* et donc la résolution des EDP. Il est important, tout d'abord, de définir quel voisinage N_{out} est nécessaire autour d'un nœud et d'une arête du réseau pour qu'ils puissent être calculés. La figure 5.3 illustre le voisinage nécessaire. Pour un nœud du réseau, le voisinage nécessaire peut être constitué des arêtes entrantes et sortantes. Pour une arête du réseau, les informations nécessaires sont les nœuds source et destination de l'arête. Par conséquent, les interfaces permettant de connaître le voisinage d'un nœud sont constituées de deux fonctions pour obtenir une liste des arêtes entrantes et sortantes, la première notée $getInEdges$, et la deuxième notée $getOutEdges$. Ces fonctions retournent un vecteur d'itérateurs sur les arêtes entrantes et sortantes. Quant aux interfaces pour le voisinage d'une arête il s'agit de deux fonctions, l'une pour connaître le nœud source, notée $getSrcNode$, et l'autre pour connaître le nœud destination, notée $getDstNode$.

Cependant ce voisinage peut être plus complexe dans certains cas. En effet, comme nous l'avons évoqué précédemment, une simulation sur les réseaux est une sous-partie des simulations multi-physiques. Cela signifie que potentiellement deux discrétisation

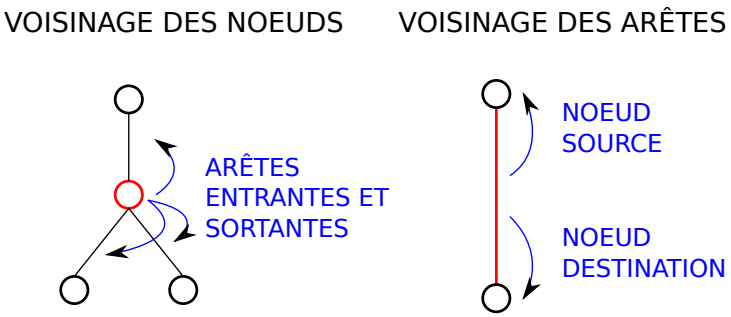


FIGURE 5.3 – *Voisinage d'un nœud et d'une arête d'un réseau.*

différentes peuvent être effectuées dans les nœuds et les arêtes afin de simuler deux phénomènes liés ensemble par un réseau. SkelGIS se limite, comme nous l'avons vu, aux discrétisations à une dimension dans les nœuds et les arêtes. SkelGIS offre donc la possibilité de définir que chaque arête ou nœud contient un tableau et non un unique élément. Cette notion est très importante pour pouvoir appliquer des schémas numériques à une dimension sur les arêtes ou les nœuds. Bien entendu, cette notion pourrait être étendue à un schéma numérique à deux dimensions et plus, mais le prototype actuel de SkelGIS n'implémente pas ces fonctionnalités. Par exemple, dans une simulation sanguine sur le réseau artériel, l'écoulement du sang dans une artère est simulé par des équations d'écoulement des fluides. Par conséquent, il faut simuler l'écoulement dans une artère par un maillage, que nous supposerons à une dimension. Dans ce cas il est important de se demander ce que devient le voisinage nécessaire pour un nœud. Ce voisinage est alors lié, tout comme pour les *DMatrix*, à l'ordre du schéma numérique appliqué dans les arêtes du réseau. La figure 5.4 illustre ce voisinage dans le cas d'un schéma d'ordre 2. En effet, dans ce cas, le nœud source de l'arête n'a pas besoin de connaître l'ensemble du maillage 1D géré par l'arête, mais uniquement ses deux premières valeurs. De même le nœud destination n'a besoin de connaître que les deux dernières valeurs du maillage de l'arête.

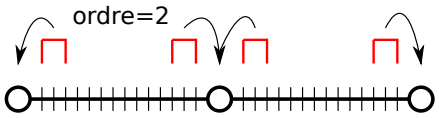


FIGURE 5.4 – *Voisinage pour le cas particulier d'un maillage 1D dans les arêtes.*

Cette notion avancée de voisinage est très importante afin de ne pas échanger des données inutiles lors des communications MPI. En d'autres termes, il est très important, pour l'efficacité du programme parallèle, que les communications MPI effectuées représentent réellement le voisinage N_{out} nécessaire. Il serait en effet dommage de communiquer tout le tableau deux fois pour chaque arête. Cette notion peut paraître compliquée, toutefois

nous allons voir dans la partie qui suit, que l'utilisation des spécialisations partielles de templates simplifie l'utilisation de ces concepts.

5.2.5 Spécialisation partielle de template

Tout comme dans le cas des maillages cartésiens à deux dimensions, le mécanisme de spécialisation de template a été utilisé pour SkelGIS dans son implémentation pour les réseaux. Une fois encore, la détermination des bons paramètres de template permet d'éviter des conditions coûteuses dans le code et permet d'éviter la mise en place du système d'héritage virtuel du C++, coûteux à l'exécution. Nous avons vu pour l'objet *DMatrix* que ses spécialisations étaient liées à la valeur de son ordre, ainsi qu'à la définition de sa connectivité. Pour les réseaux, les paramètres sont également liés au voisinage nécessaire (l'ordre de la simulation) mais également au type de données appliquées aux arêtes ou aux nœuds du réseau.

Tout d'abord, contrairement à l'objet *DMatrix* de SkelGIS, l'objet *DDAG* ne concerne que la définition du réseau et sa connectivité et non les données qui lui sont associées. Pour cette raison, conformément à la méthode SIPSim, cet objet est peu manipulé par l'utilisateur. Bien que l'objet DDAG soit responsable de l'efficacité de l'ensemble de la solution, et que l'ensemble de l'implémentation repose sur lui, les spécialisations de templates ne portent pas sur lui mais sur les données qui y sont appliquées, les DPMap. De cette manière, les choix effectués pour les voisinages ne sont pas figés pour toutes les données de la simulation, ce qui laisse plus de liberté à l'utilisateur. Comme nous l'avons vu précédemment, certaines quantités à simuler peuvent en effet participer au schéma explicite (2.3) par le calcul de $\sigma(x, t-1)$, mais pas par le calcul de $\sigma(y, t-1); y \in N(x)$. Il est donc important pour l'efficacité de la solution de laisser la notion de voisinage au niveau des quantités à simuler et non au niveau du réseau. La figure 5.5 donne la définition des objets *DPMap_ Edges* et *DPMap_ Nodes*. Le premier paramètre de template, appelé T, indique le type de données à appliquer sur le réseau. Le deuxième paramètre, appelé *node_ access* va donner des indications sur la participation de l'instance au calcul de $N(x)$.

```
template<class T, int node_access> struct DPMap_Edges
template<class T, int node_access> struct DPMap_Nodes
```

FIGURE 5.5 – *Définition des objets DPMap_ Edges et DPMap_ Nodes.*

Les deux paramètres T et *node_ access* sont liés dans leur spécialisation. Deux cas de spécialisation se présentent pour le paramètre T, et suivant ce cas la spécialisation de *node_ access* sera différente. Le paramètre T représente le type de données qui va être appliqué au réseau. Ce peut être (1) soit un type de base du C++, comme *int*, *float*, *double* etc., (2) soit un pointeur d'un type de base, autrement dit un tableau à une dimension, comme *int**, *float**, *double** etc.

1. Dans le premier cas, cela signifie que la donnée associée à chaque nœud ou chaque arête du réseau ne contiendra qu'une unique valeur. Dans ce cas *node_ access* pourra

prendre seulement deux valeurs, 0 ou 1. Si $node\_access = 0$, alors cela signifie que la quantité instanciée n'est pas concernée par les calculs du type $N(x)$, et que cette quantité est donc uniquement utilisée localement. Dans ce cas les échanges MPI n'auront pas lieu sur cette quantité ce qui améliorera les performances. À l'inverse, si $node\_access = 1$, la quantité sera utilisée dans les calculs du type $N(x)$ et les échanges MPI seront effectués.

2. Dans le cas où T est un type pointeur, ce qui signifie que chaque nœud ou chaque arête du réseau est associé à un tableau de valeurs, nous aurons, $node\_access \in [\![0, n]\!]$ où n est le nombre d'éléments dans le tableau. Comme dans le cas précédent si $node\_access = 0$ alors la quantité à simuler n'interviendra pas dans les calculs du type $N(x)$ et les échanges MPI n'auront pas lieu. Si $node\_access > 0$ alors la valeur de $node\_access$ indique l'ordre du schéma numérique qui sera appliqué dans les nœuds ou dans les arêtes, et indique donc l'échange nécessaire pour effectuer le calcul. Notons que si $node\_access = 1$, il s'agit d'un cas plus simple à gérer, une spécialisation de ce cas est donc implémentée.

Les cinq spécialisations de templates nécessaires à l'objet DPMap_Edges, pour gérer ces cas, sont présentées dans la figure 5.6.

```
template<class T, int node_access> struct DPMap_Edges
template<class T> struct DPMap_Edges<T,1>
template<class T> struct DPMap_Edges<T,0>
template<class T,int node_access> struct DPMap_Edges<T*,node_access>
template<class T> struct DPMap_Edges<T*,1>
template<class T> struct DPMap_Edges<T*,0>
```

FIGURE 5.6 – *Spécialisations partielles de template pour l'objet DPMap_Edges*

Les notions complexes de voisinage présentées dans la section précédente sont donc entièrement gérées grâce à deux paramètres de template spécialisés. L'utilisateur n'a qu'à se soucier du type d'éléments dans les nœuds et les arêtes et de l'ordre du schéma numérique 1D appliqué, pour que l'ensemble soit optimisé en terme de communications MPI.

5.3 Structure de données distribuée pour les réseaux

Dans la section précédente, le DDS DDAG pour les simulations sur les réseaux a été présenté. Nous détaillons dans cette partie l'implémentation de cette structure de données distribuée [44]. Comme toute *DDS* de la méthode SIPSim, cet objet est en grande partie responsable de l'efficacité de la solution. Son implémentation dérive du format *Compressed Sparse Row* (CSR) que nous allons tout d'abord décrire. L'adaptation et la parallélisation du format CSR seront ensuite présentées pour enfin expliquer comment s'articule l'implémentation générale de SkelGIS pour les réseaux autour de ce *DDS*. Notons qu'une version parallèle du format CSR a déjà été proposée, et fait partie de

la bibliothèque PBGL [51,62]. Toutefois, cette implémentation n'est pas spécifiquement développée pour les simulations scientifiques basées sur des maillages. Notre implémentation propose un certain nombre d'optimisations propres aux réseaux et aux simulations, que nous allons décrire dans cette section.

5.3.1 Le format Compressed Sparse Row

La variation à trois tableaux du format Compressed Sparse Row (CSR) permet de stocker des matrices creuses de façon relativement légère puisqu'il permet de ne stocker que les éléments qui ne sont pas des zéros, aussi appelés des éléments *non-nuls* dans le reste de ce travail. Ce format est donc constitué de trois tableaux dont voici la description :

- ***values*** contient les valeurs des éléments *non-nuls* qui sont stockés ligne après ligne.
- ***columns*** est de même taille que le tableau précédent. L'élément i du tableau *columns* contient l'index de colonne associé à l'élément i du tableau *values*.
- ***rowIndex*** est le dernier tableau du format dans lequel l'élément i contient l'index, dans le tableau *values*, du premier élément non-nul de la ligne i de la matrice.

Notons qu'un premier élément factice, égal à zéro, est ajouté au début du tableau *rowIndex* de telle sorte que la ligne i contient $rowIndex[i+1]-rowIndex[i]$ éléments non-nuls. Pour pouvoir accéder à un élément non-nul, avec le format CSR, en connaissant ses index de ligne et de colonne (i, j), il faut trouver la valeur j entre les éléments $columns[rowIndex[i]]$ et $columns[rowIndex[i+1]-1]$. L'index k auquel est trouvée la valeur j représente alors l'index dans le tableau *values* où se trouve l'élément non-nul recherché. Pour cette raison, le format CSR est peu efficace pour manipuler, de façon répétée, des accès aux éléments d'une matrice creuse avec les index de lignes et de colonnes. Cependant, nous allons montrer que ce format peut être très efficace pour stocker la connectivité d'un graphe.

Un graphe non-orienté est défini par $G = (V, E)$ où V est un ensemble fini de nœuds et $E \subseteq V \times V$ est un ensemble fini d'arêtes. La matrice $Sp(G)$ associée au graphe G représente la matrice d'adjacence du graphe G, comme illustré dans la figure 5.7. Dans une matrice d'adjacence, chaque élément non-nul de la matrice représente une arête $e \in E$ du graphe G, et les index de ligne et de colonne représentent les deux nœuds aux extrémités de l'arête. Notons donc que pour un graphe non-orienté, la matrice $Sp(G)$ est symétrique.

Dans un graphe $G = (V, E)$, v_i et $v_j \in V$ sont appelés des nœuds voisins si $(v_i, v_j) \in E$. En d'autres termes, deux nœuds sont voisins si deux éléments non-nuls existent dans la matrice $Sp(G)$ aux emplacements (v_i, v_j) et (v_j, v_i). Pour tout $v \in V, N(v)$ est l'ensemble des nœuds voisins du nœud v. Le degré d'un nœud $v \in V$, noté $deg(v)$, est le nombre d'arêtes incidentes de v, ce qui signifie que $deg(v) = |N(v)|$. Dans la ligne v de la matrice $Sp(G)$, $N(v)$ représente les index des colonnes où des éléments non-nuls sont présents.

Définition 2 *Dans un graphe non-orienté $G = (V, E)$ où $V = \{v_0, ..., v_{n-1}\}$, le* degré cumulé *d'un nœud $v_i \in V$ est noté $cdeg(v_i)$ et est défini par $cdeg(v_i) = \sum_{j=0}^{i} deg(v_j)$.*

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	1	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	1	0	0	0	1	1	1
5	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	1	0	0	0

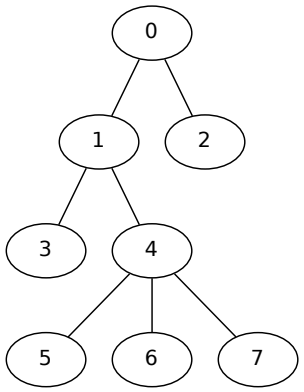


FIGURE 5.7 – *Graphe non orienté G et sa matrice d'adjacence $Sp(G)$.*

Dans la matrice $Sp(G)$, $cdeg(v_i)$ représente le nombre d'éléments non-nuls de la ligne v_i additionné au nombre d'éléments non-nuls des lignes précédentes. Il est donc possible de représenter G avec deux tableaux :

- Le premier, de taille $n+1 = |V|+1$, et appelé $cdeg$, est défini par $cdeg[i+1] = cdeg(v_i), \forall i \in [\![0, n[\![$, où $cdeg[0] = cdeg(v_{-1}) \stackrel{def}{=} 0$.
- Le deuxième, appelé N est de taille $cdeg[n] = cdeg(v_{n-1})$ et est défini pour tout $v_i \in V$ par, $N(v_i) = \{v_{N[j]} | j \in [cdeg[i], cdeg[i+1][\}$.

Cette représentation, en utilisant deux tableaux pour stocker un graphe G, correspond en fait au format CSR pour la matrice $Sp(G)$. En effet, les tableaux $cdeg$ et N de G correspondent aux tableaux *rowIndex* et *columns* de $Sp(G)$. La figure 5.7 nous donne un graphe et sa matrice d'adjacence. Dans ce graphe le nœud 0 a deux nœuds voisins, 1 et 2, nous avons alors la deuxième valeur du tableau $cdeg$ égale à 2, et les deux premières valeurs de N égales à 1 et 2. Le nœud 1 a ensuite trois voisins, la troisième valeur de $cdeg$ est alors $cdeg[2] = 2+3 = 5$. En procédant ainsi pour chaque nœud du graphe, nous obtenons, $cdeg = [0, 2, 5, 6, 7, 11, 12, 13, 14]$ et $N = [1, 2, 0, 3, 4, 0, 1, 1, 5, 6, 7, 4, 4, 4]$. Grâce à cette représentation du graphe avec deux tableaux, il est possible, par exemple, d'accéder très facilement au voisinage du nœud 4. En effet, $cdeg[4] = 7$ et $cdeg[5]-1 = 10$ indiquent le premier et le dernier index de N où se trouvent les voisins du nœud 4. Les nœuds voisins du nœud 4 sont donc les nœuds 1, 5, 6 et 7. De façon plus générale, supposons que les données associées à chaque nœud soient stockées dans un troisième tableau X tel que $|X| = |V|$. Alors, pour accéder aux valeurs voisines d'un nœud v_i, il suffit d'accéder aux éléments $N[cdeg[i]]$ à $N[cdeg[i+1]-1]$ dans X.

Le format CSR, initialement fait pour stocker des matrices creuses, est donc un format très intéressant pour stocker la connectivité d'un graphe et pour retrouver en $O(1)$ les voisins d'un nœud du graphe.

5.3.2 Format pour les DAG distribués

Comme nous l'avons indiqué dans la section 5.1, l'implémentation actuelle de SkelGIS n'a été développée que pour le sous-cas des réseaux pouvant être représentés par des DAG (par exemple celui de la figure 5.8). Cette partie va étudier l'adaptation du format CSR, décrit dans la partie précédente, pour les DAG. Nous allons décrire comment ce format a été optimisé pour le cas des simulations scientifiques, et comment il a été parallélisé.

Un graphe orienté $G = (V, E)$ est un graphe pour lequel chaque arête $e = (v_1, v_2) \in E$ est dirigée de v_1 vers v_2, et où v_1 et v_2 sont respectivement appelés le nœud *source* et le nœud *destination* de l'arête e. Un graphe orienté acyclique (DAG) est un graphe $G = (V, E)$ orienté tel que pour tout $v \in V$, il n'y a pas de chemin, en suivant les arêtes successives, de v vers lui même.

Dans les simulations scientifiques sur les DAG, pour être calculé, un nœud peut avoir besoin de ses arêtes entrantes et de ses arêtes sortantes. De son côté, pour être calculée, une arête a uniquement besoin de son nœud source et de son nœud destination (figure 5.3). Dans un DAG $G = (V, E)$, pour un nœud $v \in V$ et une arête $e \in E$, $S(e)$ est le nœud source de e et $D(e)$ est le nœud destination de e. $N_V^+(v)$ décrit l'ensemble des nœuds sortants de $v \in V$ et est défini par $N_V^+(v) = \{v' | (v, v') \in E\}$. $N_E^+(v)$ est l'ensemble des arêtes sortantes de $v \in V$ et est défini par $N_E^+(v) = \{e \in E | S(e) = v\}$. De façon symétrique, $N_V^-(v)$ et $N_E^-(v)$ sont définis comme les ensembles de nœuds et arêtes entrantes de $v \in V$. Un nœud racine $v \in V$ d'un DAG G vérifie que $|N_E^-(v)| = 0$. À l'inverse, un nœud feuille $v \in V$ vérifie que $|N_E^+(v)| = 0$.

La définition 2 doit alors être adaptée au cas des DAG pour les éléments entrants dans un nœud et ceux sortants d'un nœud. Dans un DAG $G = (V, E)$, où $V = \{v_0, ..., v_{n-1}\}$, pour un nœud $v_i \in V$, $cdeg^+(v_i) = \sum_{j=0}^{i} |N_E^+(v_j)|$ définit le degré cumulé sortant pour le nœud v_i, $cdeg^-(v_i) = \sum_{j=0}^{i} |N_E^-(v_j)|$ définit le degré cumulé entrant pour le nœud v_i. Notons ici que les degrés cumulés ont la particularité d'être les mêmes pour les arêtes et les nœuds d'un DAG. En effet, pour un nœud $v \in V$, le nombre de nœuds entrants est égal au nombre d'arêtes entrantes. De même le nombre de nœuds sortants est égal au nombre d'arêtes sortantes : $\forall v_j \in V$, $|N_E^+(v_j)| = |N_V^+(v_j)|$ et $|N_E^-(v_j)| = |N_V^-(v_j)|$.

Comme dans la partie précédente, les tableaux $cdeg^+$ et $cdeg^-$, de taille $|V|+1 = n+1$, sont définis par $cdeg^+[i+1] = cdeg^+(v_i)$ et $cdeg^-[i+1] = cdeg^-(v_i)$, où $cdeg^+[0] = cdeg^-[0] = 0$. Enfin il est également possible de définir les tableaux N_E^+ et N_E^- de taille $cdeg^+[n]$ et $cdeg^-[n]$ qui représentent les index de voisinage associés aux degrés cumulés. Enfin, deux tableaux S et D de taille $|E|$ représentent l'ensemble des nœuds sources et destinations pour chaque arête de telle sorte que $S[i] = S(e_i)$ et $D[i] = D(e_i), \forall e_i \in E$.

La figure 5.8 représente un DAG simple que nous allons prendre comme exemple pour cette nouvelle structure de données. Le nœud 0 n'a pas de voisin entrant mais a deux voisins sortants. Par conséquent, la deuxième valeur de $cdeg^-$ est égale à 0, et la deuxième valeur de $cdeg^+$ est égale à 2. Les index des voisinages associés sont stockés dans N_E^+ et N_E^-. Nous obtenons les tableaux suivants :

$$cdeg^+ = [0, 2, 4, 4, 4, 7, 7, 7, 7],\ cdeg^- = [0, 0, 1, 2, 3, 4, 5, 6, 7]$$

$$N_E^+ = [0, 1, 2, 3, 4, 5, 6],\ N_E^- = [0, 1, 2, 3, 4, 5, 6]$$

$$S = [0, 0, 1, 1, 4, 4, 4],\ D = [1, 2, 3, 4, 5, 6, 7]$$

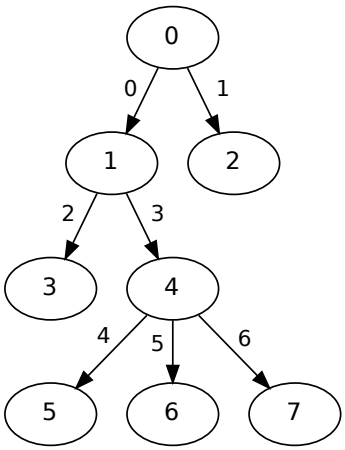


FIGURE 5.8 – *Graphe orienté acyclique correspondant au graphe 5.7.*

La structure de données ainsi obtenue est adaptée aux DAG. En revanche, cette structure de données ne comporte aucune optimisation spécifique pour les simulations scientifiques et n'est pas non plus encore une structure de données distribuée. Nous allons décrire ces deux points dans le reste de cette partie. Notons que cette section ne s'intéresse pas au partitionnement d'un graphe orienté acyclique, que nous verrons dans la section 5.5, mais uniquement à la structure de données permettant de stocker les informations locales à chaque processeur de façon efficace.

Une fois le partitionnement de graphe effectué, chaque processeur reçoit donc une partie du graphe de départ avec ses index globaux. Étant donné que la structure de données, vue précédemment, repose sur une indexation contiguë commençant à 0 pour les tableaux, cela implique tout d'abord qu'une ré-indexation locale va être nécessaire pour la version distribuée de la structure. La figure 5.9 montre un graphe global partitionné pour quatre processeurs. Nous nous intéressons à la partie bleue de ce graphe global, qui va être distribuée au processeur 1. Ce sous-graphe $G_1 = (V_1, E_1)$ possède $|V_1| = 8$ nœuds et $|E_1| = 7$ arêtes. La partie bleue de ce DAG n'est toutefois pas la seule partie à laquelle devra s'intéresser le processeur 1. En effet, pour gérer comme il se doit les voisinages des nœuds et des arêtes, des informations supplémentaires, provenant d'autres processeurs, seront nécessaires. La figure 5.10(a) montre les nœuds et arêtes dont le processeur 1 aura besoin pour effectuer convenablement ses calculs. On y retrouve la partie bleue du graphe global mais également des nœuds et arêtes additionnels en pointillés. Ces nœuds et ces arêtes sont des données qui ne sont pas locales au processeur 1 mais dont le processeur 1 aura besoin pour effectuer ses calculs. Ces données devront donc être insérées dans la structure pour permettre les calculs et être accessibles efficacement pour assurer de bonnes performances.

Le premier point auquel s'intéresser pour distribuer la structure de données est donc

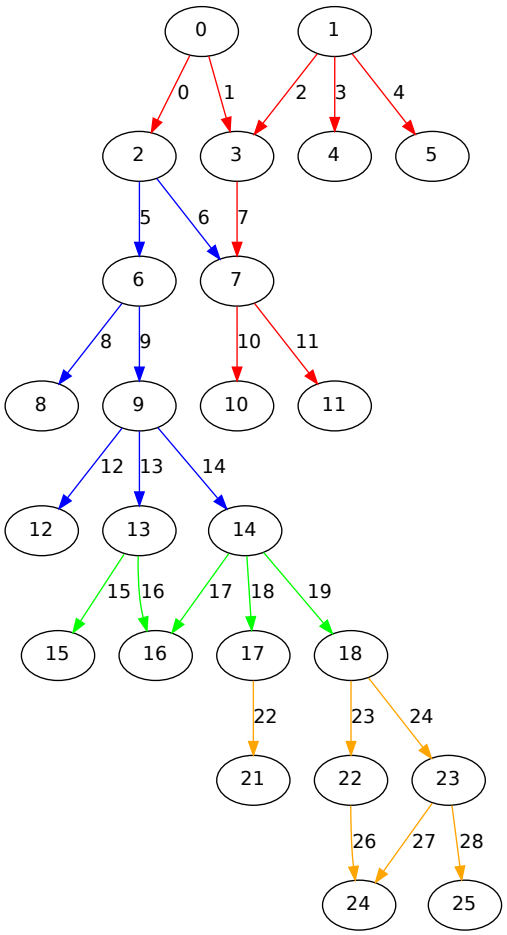
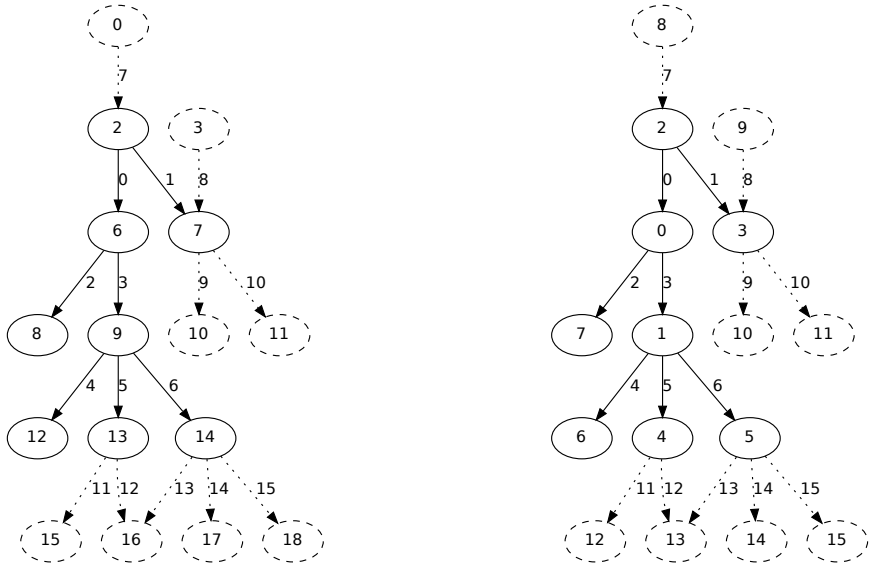


FIGURE 5.9 – *DAG global partitionné pour quatre processeurs. Le processeur 1 récupère la partie bleue de ce partitionnement.*

de ré-indexer le graphe local de chaque processeur afin de pouvoir utiliser les tableaux présentés précédemment. La ré-indexation des arêtes est la plus simple. Elle peut être quelconque. Dans la figure 5.10 la ré-indexation consiste à numéroter les arêtes de haut en bas et de gauche à droite. La ré-indexation des nœuds locaux, quant à elle, a été pensée afin d'optimiser l'utilisation des lignes de cache et de limiter les défauts de cache. Cette optimisation est liée au fait que l'on cherche à résoudre des simulations scientifiques et à leurs caractéristiques particulières. L'optimisation de la ré-indexation des nœuds consiste à ordonner les nœuds par classe de façon à pouvoir se déplacer dans la classe de nœuds de façon contiguë en mémoire. Quatre classes de nœuds ont été identifiées. Tout d'abord, il a été présenté dans la section 5.2 l'intérêt d'un recouvrement des communications par



(a) Partie du graphe global qui intéresse le processeur 1 pour gérer ses données locales et les voisinages nécessaires.

(b) Ré-indexation du sous-graphe géré par le processeur 1.

FIGURE 5.10 – *Sous-graphe géré par le processeur 1 avant et après ré-indexation.*

les calculs, afin de limiter le déséquilibrage de charge d'un partitionnement du domaine pour les différents processeurs. Cette technique connue du parallélisme repose sur le fait que les communications, nécessaires pour calculer les éléments au bord du domaine local, sont recouvertes par les calculs purement locaux qui peuvent d'ores et déjà être effectués. Pour cette raison les deux premières classes de nœuds de la ré-indexation sont les nœuds *locaux-internes* du domaine et les nœuds *locaux-externes* du domaine. Les premiers ne nécessitent pas de communications avec les autres processeurs pour être calculés, à l'inverse des seconds. Il a également été présenté dans les sections 2.2 et 5.1 que les simulations scientifiques, pour être plus réalistes sur le phénomène simulé, opèrent des comportements particuliers pour les éléments de la bordure physique du domaine global étudié. De plus, dans un réseau de type DAG, les nœuds de la bordure physique ne sont autres que les feuilles et les racines du DAG. Les deux autres classes de nœuds concernées par la ré-indexation sont donc les feuilles, puis les racines. Le passage de la figure 5.10(a) à la figure 5.10(b) illustre cette ré-indexation des nœuds locaux (en traits continus) en quatre classes, et la figure 5.11 résume ce système de ré-indexation.

Cette ré-indexation des éléments locaux à un processeur possède deux avantages en terme de performances. Tout d'abord, elle favorise la bonne utilisation des lignes de mémoire cache. En effet, dans une simulation scientifique sur les réseaux, toutes les racines vont être explorées afin de décrire leur comportement, de même pour les feuilles. Ces

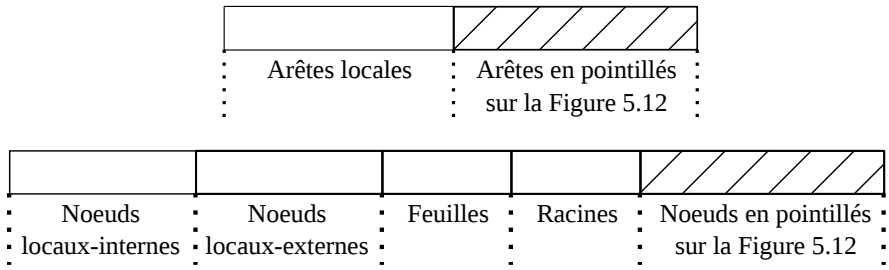


FIGURE 5.11 – *Parallélisation de la structure et ré-indexation.*

explorations d'éléments se feront donc de façon à éviter les défauts de cache en mémoire puisque la lecture entière d'une ligne de cache chargée sera favorisée avant qu'une autre ne se charge. Pour ce qui est des nœuds *locaux-internes* et *locaux-externes* le même phénomène se produit puisque lors d'un recouvrement des communications par les calculs les nœuds *locaux-internes* seront tout d'abord explorés, puis par la suite les nœuds *locaux-externes*. Le deuxième avantage de cette ré-indexation est le fait d'éviter des conditions coûteuses dans le code. En effet, une solution naïve pour les bordures physiques serait d'explorer tous les nœuds du DAG et, pour chacun, de tester si il s'agit d'une racine, d'une feuille, ou d'un nœud interne. Cela signifierait qu'à chaque itération de temps et pour chaque élément du domaine (potentiellement très grand), deux tests seraient effectués. Ces conditions seraient extrêmement coûteuses en performances. Dans cette solution les racines et les feuilles sont regroupées et les itérateurs (expliqués précédemment) seront utilisés pour naviguer dans l'une ou l'autre des classes. De même, pour le recouvrement des communications par les calculs, une solution naïve serait de tester pour chaque élément si il s'agit d'un nœud *local-interne* ou *local-externe* ce qui nuirait aux performances.

Étant donnée la ré-indexation locale du processeur 1, présentée dans la figure 5.10(b) et sans prendre en compte les nœuds et arêtes en pointillés, les six tableaux de la structure de données locale au processeur 1 sont :

$$cdeg^+ = [0,2,5,7,7,7,7,7,7],\ cdeg^- = [0,1,2,2,3,4,5,6,7]$$

$$N_E^+ = [2,3,4,5,6,0,1],\ N_E^- = [0,3,1,5,6,4,2]$$

$$S = [2,2,0,0,1,1,1],\ D = [0,3,7,1,6,4,5]$$

La ré-indexation locale permet donc d'optimiser les performances de la structure de données et permet également de pouvoir exprimer les huit tableaux locaux à chaque processeur. Toutefois il est encore nécessaire de paralléliser cette structure de données en insérant les nœuds et arêtes en pointillés. La première étape de cette parallélisation est de continuer la ré-indexation commencée avec les éléments locaux à chaque processeur, sur les éléments en pointillés. Les index des éléments en pointillés doivent être supérieurs aux index des éléments locaux. Ainsi, étant donnée une arête e_i en pointillés, et étant donnée la ré-indexation des arêtes locales $E = \{e_0, e_1, ..., e_{m-1}\}$ telle que $|E| = m$, alors

la ré-indexation de e_i est notée e_j où $j \geq m$. De même, étant donné un nœud v_i en pointillés, et étant donnée la ré-indexation des nœuds locaux $V = \{v_0, v_1, ..., v_{n-1}\}$ tel que $|V| = n$, alors la ré-indexation de v_i est notée v_j où $j \geq n$. La fin de cette ré-indexation est illustrée de la figure 5.10(a) à la figure 5.10(b) et dans la figure 5.11. En ré-indexant de cette façon les éléments en pointillés, les optimisations mises en place, pour l'utilisation des lignes de cache et les conditions, sont conservées.

Pour comprendre l'insertion des éléments pointillés dans la structure de données, continuons à utiliser l'exemple du processeur 1. Dans la figure 5.10(b), il peut être noté que les nœuds 2 et 3 doivent recevoir chacun une arête entrante du processeur 0. Pour cette raison, comme illustré dans la figure 5.12, le tableau $cdeg^-$ est modifié aux index 3 et 4 en ajoutant 1 à chacun. Comme $cdeg^-$ représente les degrés cumulés, l'addition de ces éléments doit être reportée sur tous les éléments suivants du tableau. Pour continuer cet exemple, le nœud 2 reçoit en entrée l'arête 7 du processeur 0, et le nœud 3 reçoit en entrée l'arête 8 du processeur 0. En d'autres termes, les arêtes 7 et 8 font partie du voisinage entrant des nœuds 2 et 3. Pour cette raison les index 7 et 8 doivent être insérés dans le tableau N_E^- comme illustré dans la figure 5.12. En procédant comme suit, la structure de données distribuée obtenue pour le processeur 1 est :

$$cdeg^+ = [0, 2, 5, 7, 9, 11, 14, 14, 14],\ cdeg^- = [0, 1, 2, 3, 5, 6, 7, 8, 9]$$

$$N_E^+ = [2, 3, 4, 5, 6, 0, 1, 9, 10, 11, 12, 13, 14, 15],\ N_E^- = [0, 3, 7, 1, 8, 5, 6, 4, 2]$$

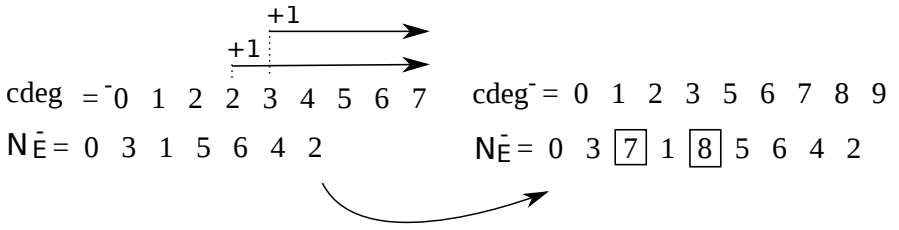


FIGURE 5.12 – *Système de ré-indexation.*

En procédant de cette façon pour paralléliser la structure de données, la caractéristique du format CSR qui permet d'accéder au voisinage d'un nœud d'un graphe en $O(1)$, est conservée pour tous les éléments de la structure, qu'ils proviennent d'autres processeurs ou pas. De plus, l'optimisation de cache mise en place est conservée.

Il reste un dernier point à aborder afin que la structure de données soit entièrement parallélisée. En effet, le travail décrit jusqu'à présent permet d'exprimer, pour chaque processeur, la connectivité de son DAG local y compris avec les éléments qui vont être communiqués par d'autres processeurs. Afin de pouvoir effectuer les communications nécessaires, une sorte de cartographie des communications est nécessaire. Grâce à elle, chaque processeur saura à qui envoyer et de qui recevoir des nœuds et des arêtes. Cette cartographie va reposer de nouveau sur les degrés cumulés. Le tableau $cdeg^{tor}$, de taille $p + 1$, où p est le nombre de processeurs, indique le nombre d'éléments à recevoir de

chacun des autres processeurs de façon cumulée. À l'inverse, le tableau $cdeg^{tos}$, de taille $p+1$ indique le nombre d'éléments à envoyer à chacun des autres processeurs, de façon cumulée là encore. Les tableaux complémentaires N_E^{tor}, N_E^{tos}, N_V^{tor} et N_V^{tos} indiquent les index des éléments à recevoir et à envoyer en suivant les tableaux de degrés cumulatifs, comme cela est le cas pour les tableaux déjà détaillés.

Étant donné un DAG $G = (V, E)$ partitionné en p sous-graphes $G_i = (V_i, E_i)$, $i \in [0, p[$, tel que $V = \bigcup_i V_i$ et $E = \bigcup_i E_i$, alors la table 5.1 indique la taille des tableaux locaux à chaque processeur mis en place dans le DDS DDAG présenté.

Tableau	**Taille**
$cdeg^+$	$\|V_i\|$
$cdeg^-$	$\|V_i\|$
S	$\|E_i\|$
D	$\|E_i\|$
$N_E^+ + N_E^-$	$\sum_{j=0}^{\|V_i\|} deg(v_j)$
$cdeg^{tor}$	p
$cdeg^{tos}$	p
N_E^{tor}	$cdeg^{tor}[p]$
N_V^{tor}	$cdeg^{tor}[p]$
N_E^{tos}	$cdeg^{tos}[p]$
N_V^{tos}	$cdeg^{tos}[p]$

TABLE 5.1 – *Taille des tableaux du DDS DDAG.*

La taille totale représentée par cette structure de donnée, pour chaque processeur, est donc

$$Taille = L + P + Conn + Comm, \tag{5.3}$$

avec

$$L = 2 \times |V_i| + 2 \times |E_i|,$$

$$P = 2 \times p,$$

$$Conn = \sum_{j=0}^{|V_i|} deg(v_j),$$

$$Comm = 2 \times cdeg^{tor}[p] + 2 \times cdeg^{tos}[p].$$

La taille du DDS DDAG pour chaque processeur dépend donc du partitionnement du réseau pour les opérandes L et $Comm$, mais également de la connectivité du réseau pour l'opérande $Conn$, et enfin du nombre de processeurs utilisés pour l'opérande P de l'équation (5.3). La structure de données présentée ici peut fonctionner quel que soit le partitionnement choisi au préalable. Toutefois, nous pouvons très bien noter que le partitionnement va impacter la taille de la structure de donnée locale à chaque processeur. Un bon partitionnement est donc aussi important que l'implémentation de la structure de

données afin d'obtenir de bonnes performances pour la solution de parallélisme implicite. Le partitionnement implémenté dans SkelGIS, ainsi que deux études de partitionnements supplémentaires seront détaillés dans la section 5.5.

La suite de cette section permet d'expliquer comment est implémenté chaque composant de la méthode SIPSim dans SkelGIS pour les réseaux, en fonction de l'implémentation de la DDS DDAG qui vient d'être détaillée.

5.3.3 Implémentation de SkelGIS pour les réseaux

Nous venons de présenter en détails l'implémentation du DDS DDAG de SkelGIS. Comme cela a été décrit dans le chapitre 3, le DDS est l'objet principal de la solution et tout le reste de l'implémentation repose sur lui. Dans cette section, nous allons expliquer comment chaque composant de la méthode SIPSim utilise l'objet DDAG et sa structure de données.

Dans la méthode SIPSim, un DPMap permet d'appliquer des données sur un DDS dans un objet léger. Comme nous l'avons vu, dans un réseau, deux DPMap sont nécessaires, DPMap_Nodes et DPMap_Edges. L'objet DPMap peut alors être comparé au tableau *values* du format CSR. Les objets DPMap_Nodes et DPMap_Edges stockent leur données dans un simple tableau à une dimension, ce qui permet d'obtenir des objets très légers. Ce tableau à une dimension, dont les indices vont suivre la ré-indexation de l'objet DDAG, vont donc représenter les tableaux à une dimension de la figure 5.11. Pour reprendre l'exemple de la figure 5.10, pour le processeur 1, l'élément 4 du tableau du DPMap_Nodes correspondra donc à une valeur sur le nœud 13 du DAG général. Un DPMap est donc lié à une instance de DDAG à sa création et se servira de cette instance de façon systématique dans son implémentation et pour ses interfaces. L'utilisateur va instancier un certain nombre de DPMap, autant qu'il a besoin de quantités sur le réseau pour sa simulation.

C'est à partir des instances de DPMap que l'utilisateur va accéder aux interfaces de programmation afin de coder sa simulation. La première d'entre elles est l'interface d'itération. Les itérateurs présents dans SkelGIS pour les réseaux ont déjà été présentés. L'ensemble de ces itérateurs va pouvoir être initialisé par le biais de deux méthodes des objets DPMap. L'une permettra de se placer au commencement de la classe des éléments et l'autre permettra de se placer à la fin de la classe des éléments. Chaque objet d'itération est ensuite un objet indépendant qui va se charger d'incrémenter le positionnement grâce à l'opérateur ++ et de le comparer grâce aux opérateurs $\leq, \geq, et ==$. L'opérateur ++ ne peut, bien entendu, être efficace que grâce à l'indexation contiguë des éléments d'une même classe qui a été mise en œuvre dans l'objet DDAG.

Enfin, le dernier composant de la méthode SIPSim implémenté dans SkelGIS est lui aussi lié à l'implémentation du DDAG. Il s'agit de l'applicateur qui a été décrit dans la section 5.2.3. Un applicateur permet d'appliquer une *opération* séquentielle, codée par l'utilisateur grâce aux instances de DPMap et aux interfaces de programmation, sur un ensemble de quantités à simuler. Un applicateur permet de cacher à l'utilisateur les échanges MPI qui se produisent avant une *opération*. L'applicateur pour les réseaux cache lui aussi les échanges nécessaires à chaque instance de DPMap, en appelant les méthodes

de communication des objets DPMap. Ces méthodes, bien entendu, utilisent directement les tableaux $cdeg^{tor}$, $cdeg^{tos}$, N_E^{tor}, N_E^{tos}, N_V^{tor} et N_V^{tos} décrits dans la section 5.3.2.

L'implémentation de la méthode SIPSim mise en place dans SkelGIS, pour le cas des réseaux, est une solution aboutie et optimisée qui cache à l'utilisateur une très grande complexité de code. Tous les composants de la méthode SIPSim dépendent de l'implémentation et de l'efficacité de l'objet DDAG et tous ces composants sont liés les uns aux autres pour fournir une solution adaptée à l'utilisateur, et efficace.

5.4 Simulation 1D d'écoulement du sang dans les artères

Dans cette section est détaillé un cas d'application réel de SkelGIS pour les réseaux. La simulation présentée étudie l'écoulement du sang dans un réseau artériel. La parallélisation de cette simulation a été effectuée en collaboration avec Pierre-Yves Lagrée, Jose-Maria Fullana et Xiaofei Wang [42]. Dans cette section nous allons tout d'abord expliquer brièvement le modèle mathématique utilisé puis les méthodes numériques appliquées afin de coder la simulation. Nous détaillerons ensuite la parallélisation de la simulation en utilisant SkelGIS et présenterons des résultats expérimentaux.

5.4.1 Simulation 1D d'écoulement du sang dans le réseau artériel

Les détails du modèle mathématique et des méthodes numériques utilisées peuvent être trouvés dans les travaux de Wang et Al [126]. Dans cette section, les informations utilisées dans cette thèse seront présentées, mais nous ne rentrerons que peu dans les détails mathématiques de la simulation.

5.4.1.1 Modèle mathématique

Le système d'équations de Navier-Stokes [73] a déjà été décrit dans la section 4.3.1, il permet d'étudier l'évolution temporelle d'un fluide dans un domaine Ω de l'espace $\mathbb{R}^3$. Ces équations peuvent donc également modéliser les écoulements sanguins dans un réseau artériel en trois dimensions. Toutefois, cette simulation est connue pour être très coûteuse en temps d'exécution et en mémoire utilisée, ce qui la rend généralement uniquement utilisable de façon locale, sur une unique artère ou sur une confluence de plusieurs artères, par exemple. Le travail de Wang et Al [126] étudie une simulation sur une unique dimension pour plusieurs raisons. Tout d'abord, cette simulation étant plus légère elle peut être exécutée sur un réseau artériel complet. Il est, de plus, possible d'envisager une simulation en temps réel pour la médecine, en couplant la légèreté de cette simulation au parallélisme. Enfin, le système 1D capture de façon intéressante le comportement des ondes sanguines provoquées par les pulsations cardiaques, ce qui donne des informations, elles aussi intéressantes, sur le système cardio-vasculaire. La simulation présentée traite le système d'équations de Navier-Stokes suivant une dimension, en intégrant ces équations sur la section de l'artère, autrement dit en moyennant une solution dans les deux

dimensions représentant la section du tube artériel. La simulation est alors représentée par deux EDP qui font le lien entre trois variables : la section de l'artère noté A, le débit volumétrique Q et la pression artérielle P toutes trois fonctions de x, la dimension spatiale, et t la dimension temporelle. Ces deux équations sont les suivantes :

$$\begin{cases} \frac{\partial A}{\partial t} + \frac{\partial Q}{\partial x} = 0 \\ \frac{\partial Q}{\partial t} + \frac{\partial}{\partial x}(\frac{Q^2}{A}) + \frac{A}{\rho}\frac{\partial P}{\partial x} = -C_f\frac{Q}{A} \end{cases} \tag{5.4}$$

où x représente l'axe longitudinal de l'artère, t représente le temps, et $-C_f\frac{Q}{A}$ représente le coefficient de frottement (avec C_f le coefficient de frottement de la paroi artérielle) et où ρ représente la densité du sang.

5.4.1.2 Résolution numérique et programmation

La simulation proposée dans les travaux de Wang et Al [126] établie une relation entre A et P, telle que $P = P_{ext} + \beta(\sqrt{A} - \sqrt{A_0}) + \nu_s\frac{\partial A}{\partial t}$, où β est le coefficient de raideur d'une artère, A_0 la section de l'artère non déformée, et P_{ext} la pression extérieure des vaisseaux, et ν_s le coefficient de viscosité du sang.

Ainsi, si nous posons

$$U = \begin{pmatrix} A \\ Q \end{pmatrix},\ F_c = \begin{pmatrix} Q \\ \frac{Q^2}{A} + \frac{\beta}{3\rho}A^{3/2} \end{pmatrix},\ F_v = \begin{pmatrix} 0 \\ -C_v\frac{\partial Q}{\partial x} \end{pmatrix},\ F = F_c + F_v$$

et

$$S = \begin{pmatrix} 0 \\ -C_f\frac{Q}{A} + \frac{A}{\rho}(\frac{\partial(\beta\sqrt{A_0})}{\partial x} - \frac{2}{3}\sqrt{A}\frac{\partial\beta}{\partial x}) \end{pmatrix}, C_v = \frac{A\nu_s}{\rho},$$

alors le système d'équations (5.4) s'écrit sous la forme :

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = S. \tag{5.5}$$

Dans cette équation il est considéré que P_{ext} est constant suivant l'axe x. Dans l'équation (5.5), U représente la variable conservative de la loi de conservation présentée à l'équation (2.9) de la section 2.2.4.2, F le flux et S est un terme supplémentaire appelé le terme source. Le problème s'apparente donc à la loi de conservation de la masse et de la quantité de mouvement à une dimension.

Les méthodes des différences finies et des volumes finis ont été utilisées dans les travaux de Wang et Al pour obtenir différents schémas numériques de la simulation et comparer les résultats obtenus. La méthode des volumes finis correspond alors à la description qui en a été faite dans la section 2.2.4.2 puisque la même forme d'équation de conservation à une dimension est traitée (hormis le terme source). Un schéma de résolution numérique explicite est donc obtenu pour la simulation au sein de chaque artère (équation (2.14)). Notons que dans cette simulation la variable U est traitée à l'ordre 2. Afin de résoudre les problèmes d'oscillations provoqués par les méthodes utilisées,

la reconstruction MUSCL (Monotonic Upwind Scheme for Conservative Law) est utilisée [126] et engendre l'utilisation de variables supplémentaires et de cinq calculs *stencils* supplémentaires. Comme dans toute résolution d'EDP, des conditions initiales et des conditions aux limites du domaine sont spécifiées pour permettre de réduire le spectre des solutions. Notons que l'EDP, à résoudre ici, simule l'écoulement du sang dans une unique artère et non sur le réseau. Le calcul des conditions aux limites pour chaque artère est donc nécessaire au calcul de l'EDP. La particularité des conditions limites de cette simulation vient du fait qu'il s'agit d'une simulation sur un réseau. Pour cette raison les conditions limites d'une artère sont liées aux conditions limites des artères auxquelles elle est connectée. Par conséquent, les conditions limites, dans cette simulation ne sont autre que les nœuds du réseau. Il s'agit donc d'un cas simple de simulation sur un réseau, où les nœuds ne représentent pas une simulation à part entière, mais simplement les conditions aux limites de chaque arête arrivant à une conjonction.

Afin de comprendre le comportement d'un nœud, prenons l'exemple d'un nœud ayant une artère mère et deux artères filles. Étant donné qu'en chaque artère les quantités A et Q sont simulées, on a alors en ce nœud l'arrivée de six conditions aux limites des artères : A_p^{n+1} et Q_p^{n+1} pour les conditions limites de l'artère mère, et A_{d1}^{n+1}, A_{d2}^{n+1}, Q_{d1}^{n+1} et Q_{d2}^{n+1} pour les conditions limites des deux artères filles. Un nœud respecte alors la loi de conservation des flux suivante

$$Q_p^{n+1} - Q_{d1}^{n+1} - Q_{d2}^{n+1} = 0 \tag{5.6}$$

et la conservation de la quantité de mouvement suivante

$$\frac{1}{2}\rho\left(\frac{Q_p^{n+1}}{A_p^{n+1}}\right)^2 + P_p^{n+1} - \frac{1}{2}\rho\left(\frac{Q_{d_i}^{n+1}}{A_{d_i}^{n+1}}\right)^2 - P_{d_i}^{n+1} = 0,\ \forall i \in \{1,2\} \tag{5.7}$$

L'algorithme 9 reprend l'algorithme 1, de la section 2.2.5, avec les spécificités de la simulation décrite ici. Il représente donc l'algorithme séquentiel de cette simulation.

5.4.2 Parallélisation avec SkelGIS

La simulation d'écoulement du sang, qui vient d'être décrite, est donc un cas particulier de simulation sur les réseaux. En effet, dans cette simulation les nœuds servent à exprimer le lien entre les conditions limites des artères du réseau. C'est l'une des façons d'utiliser un réseau, parmi d'autres. Toutefois, cette simulation se prête parfaitement à un premier cas d'application du prototype actuel de SkelGIS. Précisons, tout d'abord que dans cette simulation les types d'éléments $T1$ et $T2$ sont respectivement associés aux arêtes et aux nœuds du réseau. La simulation est alors organisée en quatre phases, comme cela a été décrit dans la section 5.1 :

1. communication des arêtes calculées à $t-1$,
2. calcul des nœuds à t,
3. communication des nœuds calculés à t,

Algorithme 9 : Algorithme séquentiel de la simulation artérielle.

```
Création ou lecture du réseau
Création ou lecture du domaine discrétisé pour chaque artère
Création des variables appliquées sur le réseau (artères et nœuds)
Initialisation des variables
Définition du pas de temps : t
Définition du temps maximal : tmax
while t < tmax do
    calcul des conditions limites du réseau (racines et feuilles) pour t
    for chaque nœud du réseau do
        calcul des schémas (5.4) et (5.5)
    end
    for chaque artère du réseau do
        for chaque élément du maillage x do
            calcul du schéma numérique (5.3) pour t et x
        end
    end
end
```

4. calcul des arêtes à t.

La description de la simulation nous offre ensuite toutes les clés permettant de coder cette simulation en utilisant SkelGIS. Tout d'abord cette simulation n'instancie qu'un unique réseau qui représente le réseau artériel à étudier. Une unique instanciation de l'objet *DDAG* est donc nécessaire. Il est ensuite possible d'identifier les variables principales de la simulation, à savoir A et Q, qui représentent des données appliquées sur les arêtes du réseau. Elles nécessitent donc deux instanciations de l'objet *DPMap_Edges*. Au sein d'une arête, une discrétisation suivant une dimension est effectuée et un schéma numérique d'ordre 2 est appliqué à chaque itération de temps. Pour cette raison, le paramètre de template T est pour les deux variables *double**, et le paramètre *node_access* est égal à 2 comme illustré dans la figure 5.13. Afin d'effectuer les calculs de condition aux limites

```
DPMap_Edges<double*,2> A
DPMap_Edges<double*,2> Q
```

FIGURE 5.13 – *Déclaration des variables A et Q*

sur les nœuds du réseau, l'instanciation d'une donnée plaquée sur les nœuds est ensuite nécessaire, même si les nœuds ne lisent et n'écrivent que des données sur les arêtes. La figure 5.14 illustre cette instanciation qui est très simple. Aucune discrétisation n'est effectuée au sein des nœuds du réseau (équations (5.6) et (5.7)). D'autres variables, qui n'ont pas été détaillées sont nécessaires au codage de la simulation, notamment pour la méthode de reconstruction MUSCL évoquée précédemment. Le principe reste cependant le même pour toutes les variables de la simulation appliquées aux arêtes ou aux nœuds.

```
DPMap_Nodes<double,0> nd
```

FIGURE 5.14 – *Déclaration d'une variable nd sur les nœuds du réseau*

Enfin, outre les instanciations de variables, les paramètres qui caractérisent les arêtes et les nœuds du réseau doivent être instanciés. Dans ce cas, ces données sont utilisées localement et pour cette raison le *node_ access* du DPMap correspondant est positionné à 0. Une fois le réseau, les variables, et les paramètres de la simulation instanciés, les itérations de temps peuvent commencer ainsi que la résolution des schémas numériques. L'algorithme 10 illustre la fonction *main* codée par l'utilisateur. Cette fonction reste très proche de la version séquentielle, à l'exception près qu'elle définit les objets SkelGIS qui viennent d'être évoqués.

Algorithme 10 : Fonction main codée par l'utilisateur

Création ou lecture du réseau $DDAG$
Création des variables appliquées sur le réseau (artères et nœuds) :
$DPMap\_Edges < double*, 2 > A$
$DPMap\_Edges < double*, 2 > Q$
$DPMap\_Nodes < double, 0 > nd$
...
Initialisation des variables de paramètres
Définition du pas de temps : t
Définition du temps maximal : $tmax$
while $t < tmax$ **do**
  | *apply_listi*({A,Q,etc.},{nd,etc.},bloodflow1,bloodflow2)
end

On peut observer que la différence entre cette fonction principale et sa version séquentielle est d'appeler, dans la boucle d'itérations en temps, l'applicateur *apply_ listi*. Comme cela a déjà été expliqué dans la section 5.2, l'applicateur *apply_ listi* permet d'appliquer des schémas numériques implicites en quatre étapes. Cette simulation convient donc à l'utilisation de cet applicateur. Cet appel est très important puisqu'il est en charge de cacher à l'utilisateur les échanges nécessaires aux calculs dans les phases 1 et 3 de la simulation. Les opérations *bloodflow1* et *bloodflow2*, données en paramètres de l'applicateur, sont les fonctions séquentielles utilisateur qui contiennent le code de calcul des phases 2 et 4 décrites précédemment.

Les algorithmes 11 et 12 illustrent les opérations *bloodflow1* et *bloodflow2*. Les structures de ces opérations sont, là encore, très proches de la version séquentielle. La seule contrainte imposée à l'utilisateur, tout comme dans la fonction *main*, est d'utiliser les notions propres à SkelGIS pour manipuler les instances des objets DPMap_Edges et DPMap_Nodes : les itérateurs ; les accesseurs ; et les fonctions de voisinages. La première opération, décrite dans l'algorithme 11 représente donc le code de la phase 2 de la simulation. Ce code consiste, tout d'abord à calculer les conditions limites du

réseau, à savoir les racines et les feuilles. Pour cela, les itérateurs spécifiques à ces deux classes d'éléments sont utilisés ainsi que l'opérateur [] et les fonctions de voisinage d'un nœud. Par la suite, les conditions aux limites des artères sont calculées. Ces calculs se produisent aux points de conjonction (nœuds), et sont effectués en fonction des résultats obtenus à l'itération précédente sur les artères qui se rencontrent en ce nœud. Les autres nœuds du réseau sont donc ensuite calculés dans l'opération en appliquant les schémas numériques (5.6) et (5.7). La deuxième opération, décrite dans l'algorithme 12 repré-

Algorithme 11 : Opération bloodflow1 décrivant les calculs effectués à chaque itération de temps sur les nœuds.

```
Data : {DPMap}
Result : Modification de {DPMap}
ItR := itérateur de début sur les racines
endItR := itérateur de fin sur les racines
while ItR≤endItR do
    Application des conditions limites pour les racines avec : A[ItR], Q[ItR],
    nd[ItR], A.getInEdges(ItR), Q.getOutEdges(ItR) ...
    ItR++
end
ItL := itérateur de début sur les feuilles
endItL := itérateur de fin sur les feuilles
while ItL≤endItL do
    Application des conditions limites pour les feuilles avec : A[ItL], Q[ItL],
    nd[ItL], A.getInEdges(ItL), Q.getOutEdges(ItL) ...
    ItL++
end
ItC= itérateur de début sur les nœuds
endItC := itérateur de fin sur les nœuds
while ItC≤endItC do
    Application des schémas numériques des nœuds (5.4) et (5.5) avec : A[ItC],
    Q[ItC], nd[ItC], A.getInEdges(ItC), Q.getOutEdges(ItC) ...
    ItC++
end
```

sente, quant à elle, le code de la phase 4 de la simulation. Ce code consiste à appliquer le schéma numérique issu de la méthode des volumes finis appliquée à l'équation (5.5). Pour cela, les itérateurs spécifiques à ces deux classes d'éléments sont utilisés ainsi que l'opérateur [] et les fonctions de voisinage d'une arête.

L'utilisation de SkelGIS pour coder cette simulation d'écoulement du sang dans les artères répond donc aux attentes annoncées, et propose une solution très proche du séquentiel, aussi bien en terme de structure de programme qu'en terme de programmation. En effet, aucune difficulté technique n'est introduite par la solution et aucun code pa-

Algorithme 12 : Opération bloodflow2 décrivant les calculs effectués à chaque itération de temps sur les arêtes.

```
Data : {DPMap}
Result : Modification de {DPMap}
ItA= itérateur de début sur les artères
endItA := itérateur de fin sur les artères
while ItA≤endItA do
    for chaque élément du maillage de l'artère x do
        calcul du schéma numérique issu de l'équation (5.4) pour t et x avec :
        A[ItA], Q[ItA], nd[ItA], A.getSrcNode(ItA), Q.getDstNode(ItA) ...
    end
    ItA++
end
```

rallèle n'a été produit par l'utilisateur. Dans la prochaine section vont être étudiés en détails les résultats expérimentaux obtenus sur cette version SkelGIS de la simulation.

5.4.3 Résultats

Dans cette section, nous allons présenter les résultats que nous avons obtenus sur la simulation d'écoulement du sang dans le réseau artériel décrite précédemment, et codée avec la bibliothèque SkelGIS. Nous appellerons cette simulation *bloodflow-SkelGIS*. Les résultats expérimentaux se découpent en trois parties. Tout d'abord, l'efficacité du recouvrement des communications par les calculs a été évaluée. Cette optimisation du code parallèle est, en effet, possible grâce à l'indexation mise en place dans la structure de données. Nous avons décrit ce point dans les sections 5.2 et 5.3.2. Les expériences suivantes utilisent toutes le recouvrement des communications par les calculs. L'implémentation bloodflow-SkelGIS sera ensuite comparée à une version OpenMP de la même simulation, que nous noterons *bloodflow-OpenMP*. Trois comparaisons seront effectuées, les temps d'exécution, les accélérations des programmes ainsi que les métriques de Halstead [65]. Enfin, une dernière phase d'expérimentation donnera les temps d'exécution et les accélérations de bloodflow-SkelGIS sur deux grappes du *top500* international de novembre 2013, avec des tailles variables de réseaux. Les machines utilisées dans l'ensemble de ces expériences sont détaillées dans la table 5.2. Il y a tout d'abord la grappe "Babbage", de l'Institut Jean le Rond d'Alembert UMR CNRS Université de Paris 6, qui est une grappe de taille moyenne mais bien équipée, les nœuds *thin* du super-calculateur TGCC-Curie classé vingtième dans le *top500* de novembre 2013, et enfin les nœuds IBM Blue Gene/Q de super-calculateur Juqueen en Allemagne classé huitième sur la même liste. Notons que Juqueen nous permet également de valider SkelGIS sur une architecture très différente des autres clusters. Ces machines nous ont permis de procéder à des tests de performances allant de 8 à 8192 cœurs.

Afin de détailler au mieux les expériences effectuées, notons que la simulation bloodflow-SkelGIS est exclusivement composée d'opérations sur des variables à double-

Calculateur	**Babbage**	**TGCC Curie**	**Juqueen**
Processeur	2×Intel Xeon (3 GHz)	2×SandyBridge (2.7 GHz)	IBM PowerPC (1.6 GHz)
Cœurs/nœud	12	16	16
Mémoire/nœud	24 GB	64 GB	16 GB
Compilateur [-O3]	OpenMPI	Bullxmpi	MPICH2
Réseau	Infiniband	Infiniband	5D Torus 40 GBps

TABLE 5.2 – *Spécifications matérielles des machines utilisées*

précision et que chaque expérience présentée dans cette partie a été lancée quatre fois chacune, puis moyennée. Enfin, l'écart type noté sur l'ensemble des exécutions de la simulation est très faible, et inférieur à 2%.

5.4.3.1 Recouvrement des communications par les calculs

La première expérience effectuée est celle qui permet de valider l'optimisation de recouvrement des communications par les calculs. Cette expérience a été menée sur le cluster "Babbage" de l'Université de Paris 6, sur un réseau de type arbre contenant 15.000 arêtes et 15.000 nœuds et un degré maximal très faible de 3. La table 5.3 donne l'ensemble des temps d'exécution obtenus de 16 à 384 processeurs utilisés, et la figure 5.15 illustre, de son côté, l'accélération de la simulation bloodflow-SkelGIS avec et sans l'optimisation de recouvrement.

Cœurs	Sans recouvrement	Avec recouvrement	Gain de temps %
16	12715.9	12386.2	2.59
32	6462.38	6189.79	4.22
64	3491.87	3124.74	10.5
128	1912.89	1581.46	17.3
256	1225.55	843.103	31.2
384	1166.18	612.787	47.45

TABLE 5.3 – *Temps d'exécution en secondes de bloodflow-SkelGIS avec et sans l'optimisation de recouvrement des communications par les calculs.*

Tout d'abord, l'optimisation mise en place est efficace, comme cela pouvait être attendu. On note, en effet, une différence très nette, à la fois pour les temps d'exécution et sur les accélérations, entre la simulation sans le recouvrement et la simulation avec le recouvrement. Cependant, il paraît presque surprenant d'obtenir des gains de performance aussi importants. La version sans recouvrement fait émerger une faiblesse du prototype SkelGIS. En effet, même si de meilleures performances étaient attendues avec cette optimisation de recouvrement, on observe que le speedup de la version sans recouvrement devient moins bon à partir de 128 processeurs. Cette faiblesse n'a pas été étudiée avec précision. Nous pensons que le problème peut venir du partitionnement de graphes mis

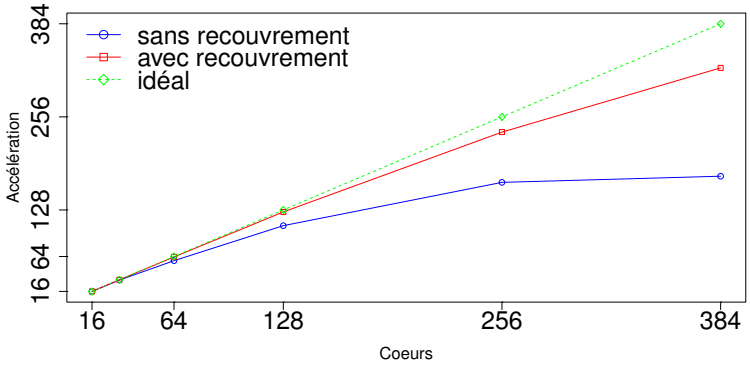


FIGURE 5.15 – *Accélération de la simulation bloodflow-SkelGIS sans et avec le recouvrement des communications par les calculs.*

en place dans le prototype actuel de SkelGIS, qui sur un grand nombre de processeurs, fournit une distribution moyennement équilibrée. Les résultats de ce partitionnement seront présentés en détails dans la section 5.5.1. Ce résultat montre, quoi qu'il en soit, l'importance de l'optimisation mise en place dans la structure de données *DDAG*.

5.4.3.2 Comparaison avec OpenMP

La méthode SIPSim et son prototype implémenté SkelGIS, se proclame comme une solution de parallélisme implicite très simple d'utilisation, car adaptée aux simulations scientifiques, et proposant de très bonnes performances parallèles. De son côté, le langage OpenMP est reconnu et utilisé dans les domaines scientifiques de par sa simplicité d'utilisation, tout en sachant que les performances obtenues ne sont pas nécessairement excellentes. Une comparaison des deux approches paraît donc être une bonne expérience, aussi bien en terme de performances qu'en terme de difficulté de programmation.

Une version OpenMP de la simulation d'écoulement du sang dans le réseau artériel a été développée par les mathématiciens à l'origine de cette simulation. La parallélisation OpenMP dite à grain fin (*fine-grain*), est la parallélisation la plus utilisée par les non-informaticiens et consiste à déclarer des boucles parallèles dans le code par la simple directive *#pragma omp parallel for*. Ces boucles sont alors automatiquement réparties entre les différents processus (threads) créés, ce qui rend la solution totalement implicite. La simulation OpenMP qui a été implémentée ici n'est toutefois pas une implémentation fine-grain. Il s'agit d'une parallélisation à gros grain (*coarse-grain*), où la directive plus générale *#pragma omp parallel* est utilisée. Ce type de parallélisation OpenMP n'est pas totalement implicite, et donc plus complexe qu'une parallélisation fine-grain. Une parallélisation coarse-grain ressemble, dans son principe, à une parallélisation MPI. En effet, une zone parallèle générale est déclarée comme entre les fonctions *MPI_ Init* et *MPI_ Finalize*. Dans cette zone parallèle, plusieurs threads sont créés et vont, sauf précision inverse de l'utilisateur, exécuter la même zone de code. Aucun transfert de données n'est nécessaire entre les processus qui partagent la même mémoire, toutefois des syn-

chronisations sont implicitement effectuées entre les processus pour garantir la cohérence des données et pour contrôler leur accès. Ce type de parallélisation permet de mettre en place des programmes parallèles en suivant le paradigme SPMD et le parallélisme de données, tout comme le modèle PGAS le propose. Ce type de parallélisation obtient dans les cas complexes de meilleures performances que la parallélisation fine-grain, mais n'est pas totalement implicite aux yeux de l'utilisateur [109]. En effet, étant donné que tous les threads exécutent le même code, il est nécessaire de gérer une distribution des données entre les threads. Cette distribution est à la charge de l'utilisateur ce qui rend la solution plus complexe à mettre en œuvre. De plus, une réflexion est nécessaire pour déclarer les variables partagées et locales. Cette parallélisation ayant été effectuée par des mathématiciens, elle donne un aperçu des performances qu'il est possible d'obtenir en utilisant OpenMP sans connaissances poussées en informatique.

La table 5.4 indique les temps d'exécution obtenus sur un unique nœud *thin* du super-calculateur TGCC-Curie. Comme indiqué dans la table 5.2, un nœud contient 16 cœurs. La figure 5.16 trace les temps d'exécution obtenus en fonction du nombre de cœurs utilisés, avec une échelle logarithmique, ce qui permet de mieux apprécier cette comparaison.

Cœurs	OpenMP	SkelGIS	Gain de temps %
1	2209.77	2006.92	9.2
2	2068.33	959.095	53.6
4	1122.73	497.424	55.7
8	621.72	273.011	56
16	341.95	156.59	54.2

TABLE 5.4 – *Temps d'exécution en secondes de bloodflow-OpenMP et bloodflow-SkelGIS.*

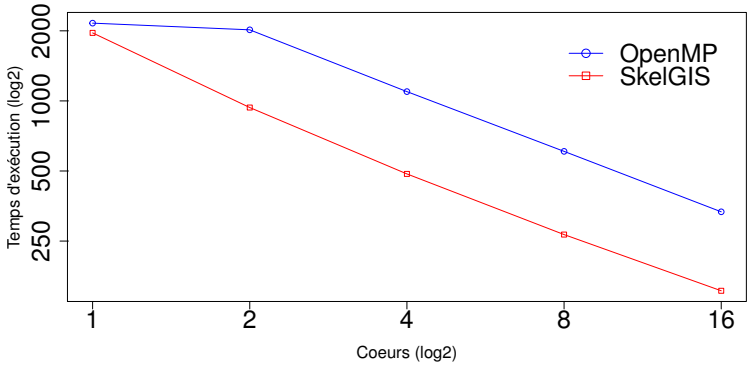


FIGURE 5.16 – *Comparaison des temps d'exécution entre bloodflow-OpenMP et bloodflow-SkelGIS avec une échelle logarithmique.*

La table 5.4 et la figure 5.16 montrent que la version OpenMP obtient un très mauvais temps d'exécution avec deux cœurs, et que ce problème se répercute, par conséquent, sur

le reste des temps d'exécution. La figure 5.17 montre l'accélération des simulations. On peut alors voir que le speedup obtenu par la version OpenMP n'est pas mauvais, car proche du linéaire, outre le pallier entre 1 et 2 cœurs. La version SkelGIS, de son côté, obtient de très bonnes performances. Le temps d'exécution séquentiel est tout d'abord meilleur que le temps séquentiel de la version OpenMP de 9%. De plus, l'accélération du programme étant quasi linéaire, et proche de l'idéal (figure 5.17), cette performance est conservée en augmentant le nombre de processeurs. Le résultat de la version OpenMP nuit à une bonne comparaison des deux approches. Cependant, notons que la pente de l'accélération de la version OpenMP est moins importante que celle de la version SkelGIS. Pour cette raison, si la version OpenMP était retravaillée afin de résoudre le pallier à 2 cœurs, les temps d'exécution et les accélérations obtenus seraient toujours moins performants pour la version OpenMP.

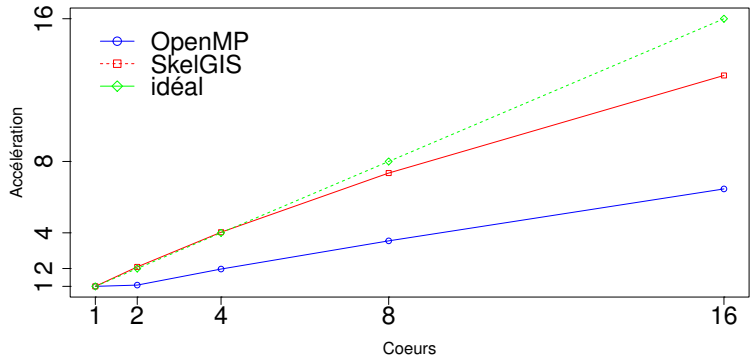


FIGURE 5.17 – *Comparaison des accélérations entre bloodflow-OpenMP et bloodflow-SkelGIS.*

Mais le point de la comparaison n'est pas réellement ici. En effet, qu'une version OpenMP meilleure que la version SkelGIS n'existe ou pas, la version SkelGIS obtient des performances très intéressantes et qui pourraient elles aussi être améliorées. En plus des performances, il paraît intéressant de comparer la difficulté de programmation des deux programmes. Pour expérimenter ce dernier point, les métriques de Halstead ont une nouvelle fois été utilisées pour obtenir une évaluation du volume du programme écrit, de la difficulté de programmation, et de l'effort à fournir. La table 5.5 donne les métriques calculées pour chacune des deux simulations : bloodflow-OpenMP et bloodflow-SkelGIS.

L'effort à fournir pour écrire la version SkelGIS est environ 45% moins grand que l'effort à fournir pour écrire la version OpenMP coarse-grain. Pourtant un programme OpenMP ne consiste qu'en un ensemble de directives de compilation à appliquer à son code, alors pourquoi un tel écart ? Comme le montre le tableau des métriques, cet écart vient principalement d'un facteur, celui du volume du programme écrit qui est 30% plus important dans la version OpenMP. Il s'agit d'un point très important pour la méthode SIPSim et pour SkelGIS. En effet, SkelGIS gère, par l'intermédiaire de l'objet *DDAG*, l'ensemble de la structure de données de façon transparente pour l'utilisateur. Ce point simplifie énormément le code utilisateur puisque celui-ci n'a plus à se soucier de coder une structure de données, qui peut être complexe suivant les cas. Pour les *DMatrix*, cet

Métriques	OpenMP	SkelGIS	Gain %
N_1	2607	2692	-3.2
N_2	10737	7424	30.8
n_1	365	231	36.7
n_2	501	282	43.7
V	130213.73	91072.48	30
D	3911.18	3040.68	22.2
E	509.3M	276.9M	45.6

TABLE 5.5 – *Métriques de Halstead pour les versions bloodflow-OpenMP et bloodflow-SkelGIS.*

avantage était moins évident puisqu'une matrice est une structure de données très simple à coder. Dans le cas des réseaux, l'intérêt du composant *DDS*, de la méthode SIPSim, prend tout son sens et simplifie de façon significative le code de la simulation. La méthode SIPSim étant une solution spécifique de parallélisme implicite en comparaison de OpenMP, ce résultat devait être observé. Cependant, notons qu'un autre phénomène apporte du poids au volume de code à fournir pour la version OpenMP. En effet, le fait que la version OpenMP étudiée produise une parallélisation de type coarse-grain ajoute du travail à l'utilisateur. Comme nous l'avons déjà décrit, cette version de parallélisation OpenMP est proche d'une parallélisation MPI et l'utilisateur doit procéder à la décomposition du domaine par lui-même. Cela nuit au parallélisme implicite, mais cela nuit également au volume de code à fournir.

La simulation bloodflow-SkelGIS est donc plus intéressante en tout point par rapport à la version bloodflow-OpenMP. La version SkelGIS obtient, en effet, de très bonnes performances et de très bonnes métriques de Halstead.

5.4.3.3 Performances de SkelGIS

La méthode SIPSim et son implémentation SkelGIS sont conçues pour être exécutées sur des architectures à mémoire distribuée. Bien qu'un programme parallèle MPI fonctionne très convenablement sur une architecture à mémoire partagée, comme nous venons de le voir, ce n'est pas son objectif premier. Cette nouvelle série d'expériences permet donc d'estimer les performances de la bibliothèque SkelGIS sur des données de taille importante et variable et sur des machines de configuration et de taille variables. La table 5.6 référence les expériences qui ont été menées.

	Nombre de nœuds et d'arêtes	calculateur utilisé
Expérience 1	15k	TGCC-Curie
Expérience 2	50k	Juqueen
Expérience 3	100k	Juqueen
Expérience 4	500k	Juqueen

TABLE 5.6 – *Expériences de performance sur bloodflow-SkelGIS.*

Les temps d'exécution obtenus pour l'ensemble de ces expériences sont réunis dans la table 5.7. Le nombre de cœurs utilisés pour chaque expérience n'est pas le même pour plusieurs raisons. Tout d'abord, la longueur des traitements et le nombre d'heures qui nous étaient allouées sur chaque calculateur ne nous permettaient pas d'effectuer des expériences trop longues. Pour cette raison, les expériences sur les réseaux de taille 50k et 100k ne commencent pas à 8 cœurs, et l'expérience sur un réseau de taille 500k commence elle à 1024 cœurs. De plus, les clusters utilisés sont très demandés et le nombre d'utilisateurs est très important. Des files d'attente pour les expériences sont donc mises en place sur ces machines. Plus il y a de cœurs réservés pour une expérience, plus le délai d'attente pour que l'expérience soit lancée est long. Pour cette raison, nous n'avons pas réservé plus de 1024 cœurs sur le TGCC-Curie, et nous ne sommes pas montés au delà de 8192 cœurs pour Juqueen. Enfin, sur Juqueen notamment, des contraintes très fortes sur l'utilisation des machines sont mises en place. Le temps de calcul alloué à chaque cœur possède un minimum et un maximum à respecter, aussi nous n'avons pu descendre sous 256 cœurs pour les traitements sur les réseaux de taille 50k et 100k. Afin de mieux appréhender les résultats obtenus, les accélérations de la simulation bloodflow-SkelGIS pour chacune des expériences sont représentées dans les figures 5.18, 5.19 et 5.20.

Cœurs	15k TGCC	50k Juqueen	100k Juqueen	500k Juqueen
8	10080.1			
16	5288.56			
32	2680.21			
64	1372.12			
128	743.103			
256	416.919	12602.20	24170.50	
512	247.537	6976.98	12650.60	
1024	178.8	3869.46	7043.42	30615.20
2048		2624.44	4122.97	16239.50
4096		1254.94	2657.76	8959.82
8192				5606.09

TABLE 5.7 – *Temps d'exécution en secondes de bloodflow-SkelGIS.*

Concernant la première expérience, le même jeu de données que dans l'expérience sur le recouvrement a été utilisé. Hors, nous pouvons remarquer dans la figure 5.15 que l'accélération est meilleure que dans la figure 5.18. En d'autres termes, l'accélération de l'expérience semble meilleure sur la grappe "Babbage" que sur le TGCC-Curie. Cette différence de performances peut être due au fait que le code SkelGIS réagit mieux à la configuration matérielle du cluster "Babbage". Toutefois cette explication parait étonnante. Toutefois, notons que les performances sont tout de même très bonnes sur un réseau de taille très modeste.

Dans les expériences suivantes, la taille des réseaux est augmentée et l'architecture utilisée reste identique ce qui permet d'analyser le passage à l'échelle de SkelGIS. Il peut être observé dans les figures 5.19 et 5.20 que les accélérations obtenues sont très convain-

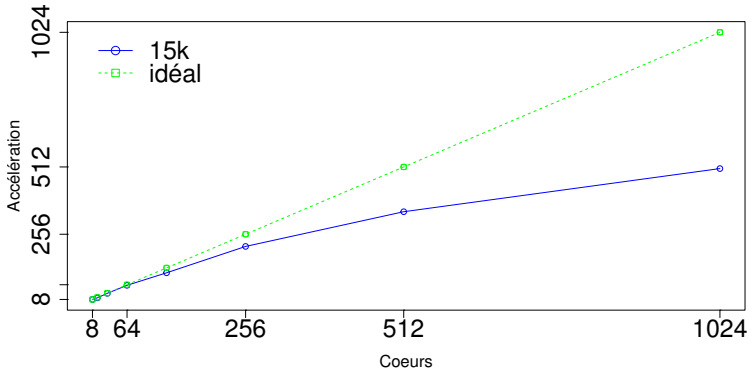


FIGURE 5.18 – *Accélération de bloodflow-SkelGIS sur un DAG de 15k arêtes et nœuds sur le TGCC.*

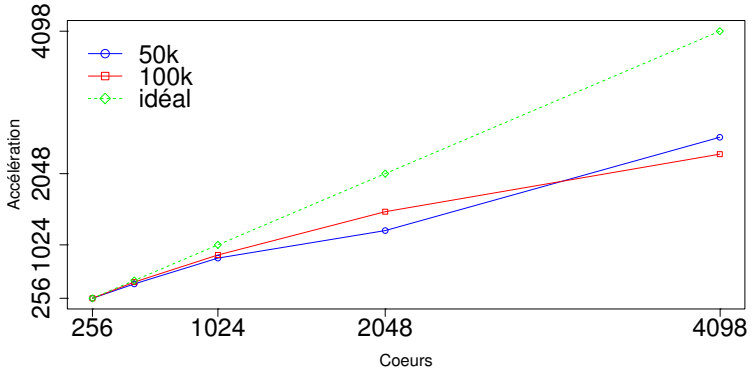


FIGURE 5.19 – *Accélération de bloodflow-SkelGIS sur des DAGs de 50k et 100k arêtes et nœuds sur Juqueen.*

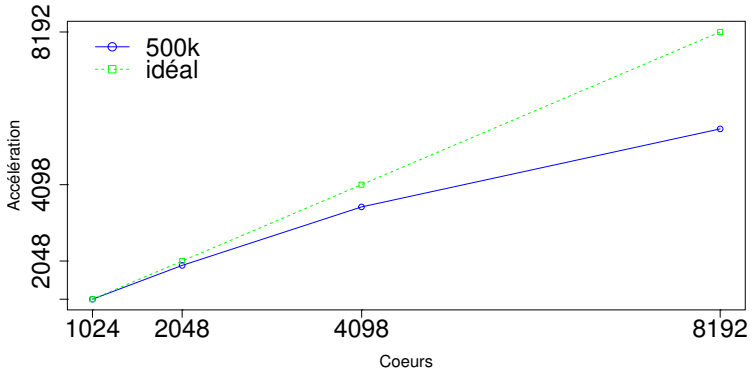


FIGURE 5.20 – *Accélération de bloodflow-SkelGIS sur un DAG de 500k arêtes et nœuds sur Juqueen.*

cantes. Le passage à l'échelle y est clairement bon puisque proche d'un résultat linéaire jusqu'à 4098 processeurs. Sur la taille de réseau la plus importante, les performances sont très bonnes jusqu'à 8192 processeurs.

Les résultats analysés dans cette section montrent que SkelGIS est convainquant en plusieurs aspects. Tout d'abord l'optimisation de recouvrement rend la solution très performante, et le passage à l'échelle est vérifié jusqu'à un très grand nombre de processeurs. De plus, en comparaison avec une version coarse-grain OpenMP, SkelGIS montre des performances très bonnes sur une architecture à mémoire partagée. OpenMP étant une référence dans le parallélisme implicite, les résultats des métriques de Halstead montrent que SkelGIS est très simple d'utilisation et qu'il peut même être plus simple qu'une version séquentielle, du fait de la structure de données implicite. Pour finir, le passage à l'échelle d'une simulation implémentée avec SkelGIS est bon et semble robuste sur différentes configurations matérielles.

5.5 Partitionnement de réseaux

Dans cette section nous allons expliquer le choix d'implémentation qui a été mis en place pour partitionner un réseau dans SkelGIS. Cette solution a montré des résultats intéressants, comme nous l'avons vu dans la section 5.4.3. Les détails de cette méthode de partitionnement seront décrits dans cette section, et nous verrons qu'elle contient un certain nombre de désavantages. Un travail récent, en collaboration avec Rob Bisseling, a permis d'étudier avec plus de précision le problème de partitionnement qui est posé par les réseaux, mais aussi la façon dont on peut résoudre ce problème en utilisant le partitionneur Mondriaan [120]. Les deux méthodes de partitionnement récemment mises en place sont également introduites et évaluées dans cette section.

5.5.1 Partitionnement par regroupement d'arêtes sœurs

Il faut tout d'abord remarquer que dans le type de simulations qui nous intéresse des calculs sont effectués à la fois sur les nœuds et sur les arêtes du graphe qui représente le réseau. Nous avons donc deux solutions pour partitionner le graphe. On peut tout d'abord choisir de distribuer les deux types d'éléments sur les processeurs, ce qui représente le véritable problème de partitionnement d'un réseau. Ce problème est toutefois complexe à résoudre, et il conduit également à une gestion des communications plus complexe. L'autre solution est alors de ne distribuer que les nœuds ou que les arêtes et de dupliquer les nœuds ou les arêtes manquantes aux coupures, afin de permettre les calculs. Dans ce cas, les communications sont moins difficiles à gérer dans la solution de parallélisme implicite. L'implémentation choisie dans le prototype actuel de SkelGIS pour les réseaux a été de partitionner les arêtes sur les processeurs, où le calcul était considéré comme plus lourd, et de dupliquer les nœuds sur les différents processeurs, aux coupures des différentes parties, où le calcul était considéré comme plus léger. Nous avons donc fait le choix, dans un premier temps, de simplifier le problème en ne partitionnant que les

arêtes du réseau et en dupliquant les nœuds. De plus, nous avons également limité, et donc modifié, le problème en considérant une sous-partie des réseaux qui peuvent être représentés par des DAG.

Comme cela a déjà été décrit dans l'état de l'art 2.3, un partitionnement de graphe découpe l'ensemble des nœuds d'un graphe $G = (V, E)$ en p parties distinctes $V_0, \ldots, V_{p-1}$ telles que pour $i \neq j$, $V_i \cap V_j = \emptyset$ et telles que $V = \bigcup_i V_i$. Le partitionnement doit répondre à la contrainte d'équilibrage de charge suivante

$$\omega(V_i) \leq (1 + \epsilon)\frac{\omega(V)}{p} \tag{5.8}$$

où ϵ est le pourcentage de non-équilibrage accepté. ϵ peut être choisi soit automatiquement par la solution, soit par l'utilisateur. Enfin, le partitionnement doit minimiser le nombre d'arêtes coupées, aussi appelé *edge-cut*.

Dans le prototype de SkelGIS, nous avons choisi de partitionner les arêtes du graphe. Le problème de partitionnement doit donc être transposé aux arêtes. Ce problème n'est pas nouveau et les travaux de Kim et Al [81] se sont, par exemple, intéressés à partitionner les arêtes en coupant les nœuds et en les dupliquant sur chaque processeur. Nous avons toutefois développé notre propre méthode de partitionnement afin d'essayer de tirer parti des DAG et du cas particulier des simulations numériques sur les réseaux. Notre méthode de partitionnement transforme tout d'abord le DAG en un graphe G' plus grossier que nous appelons méta-graphe. En reprenant les définitions introduites dans la section 5.3, nous allons décrire la construction du graphe G'. Les arêtes sortantes et entrantes d'un nœud $v \in V$ sont respectivement notées $N_E^+(v)$ et $N_E^-(v)$. Deux arêtes e_i et e_j sont considérées comme étant sœurs si $S(e_i) = S(e_j)$ et $D(e_i) \neq D(e_j)$. L'ensemble des arêtes sœurs d'un nœud $v \in V$ est défini comme égal à $N_E^+(v)$. Le méta-graphe G' du réseau G est alors défini comme le graphe $G' = (V', E')$ où $V' = \{N_E^+(v), \forall v \in V\}$ et $E' = \{(N_E^+(v_1), N_E^+(v_2)) \in V' \times V', \forall v_1 \in V, v_2 \in N_V^+(v_1)\}$. La figure 5.21 illustre le réseau de type DAG G et le méta-graphe G' associé.

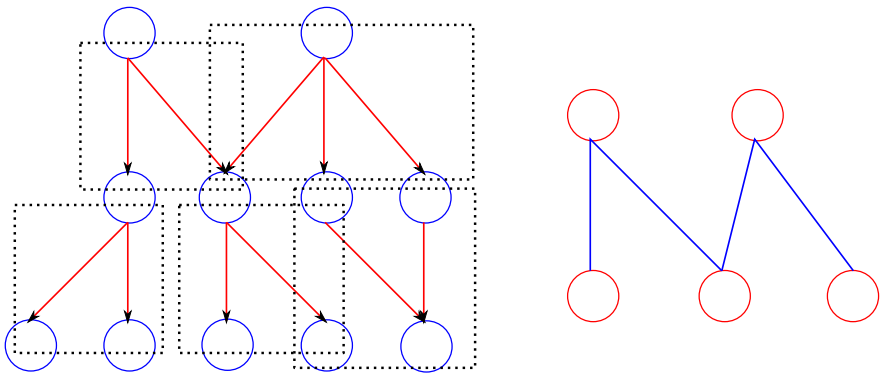


FIGURE 5.21 – *Exemple de réseau G (à gauche) de type DAG, et du méta-graphe G' associé (à droite).*

	Nombre de nœuds	**Degré maximum**
Arbre 1	10k	10
Arbre 2	10k	2
Arbre 3	16k	5

TABLE 5.8 – *Type d'arbres utilisés pour évaluer le partitionnement.*

Dans le méta-graphe G' les nœuds sont les blocs d'arêtes sœurs du graphe G, et les arêtes représentent les liens entre ces blocs par l'intermédiaire des nœuds de G. Notons que les nœuds de G' sont assortis d'un poids, qui est égal au nombre d'arêtes sœurs de G représentées par le nœud de G'. En d'autres termes, pour un nœud $v \in V$ et un nœud $v' = N_E^+(v) \in V'$, on associe un poids $\omega(v') = |N_E^+(v)|$. En partitionnant ce méta-graphe G' deux problèmes sont résolus à la fois :

— Le problème de partitionnement posé redevient un problème standard de partitionnement des nœuds d'un graphe.
— Le partitionnement des nœuds de G', en tenant compte de leur poids, revient à un partitionnement de blocs d'arêtes sœurs dans G, ce qui favorise la localité géographique du résultat.

En revenant ainsi à un problème de partitionnement standard de type *edge-cut*, il aurait été possible d'utiliser un partitionneur de graphe comme METIS [77] ou Scotch [100]. Toutefois, nous ne connaissions pas ces solutions au moment du développement de ce prototype de SkelGIS. Cette solution pourra être envisagée dans les prespectives de ce travail pour comparer avec plus de détail les différentes approches de partitionnement-possible. Nous pouvons également noter qu'en envisageant ce prototype de SkelGIS nous avons préféré une solution simple et sans dépendance pour partitionner uniquement les DAG, et non tous les graphes comme le proposent METIS et Scotch.

Afin d'évaluer les limites du partitionnement de SkelGIS, nous avons effectué quelques expériences en calculant le nombre d'arêtes dont chaque processeur a la charge, et les communications effectuées par chaque processeur, en terme de nombre d'octets et de temps effectif. Nous avons mené ces expériences sur trois types d'arbres différents, générés aléatoirement, dont la description est résumée dans la table 5.8. Le premier arbre ainsi obtenu est un arbre plutôt large et peu profond alors que le second arbre est à l'inverse un arbre profond et peu large. Enfin, le dernier arbre est un arbre qui peut être considéré comme assez équilibré en profondeur et en largeur.

La table 5.22 donne le nombre moyen d'arêtes dont chaque processeur est en charge à chaque itération de simulation, et l'écart type maximal obtenus par rapport à cette valeur moyenne. Ainsi, plus l'écart type est proche de la valeur moyenne, plus l'équilibrage de charge peut être considéré comme mauvais. On peut observer, dans cette table, que l'équilibrage de charge en terme de nombre d'arêtes est bon puisque l'écart type obtenu est très faible comparé à la valeur moyenne. On peut également observer que le partitionnement effectué est meilleur pour les arbres 2 et 3 que pour un arbre large comme l'arbre 1.

La table 5.23 représente le nombre moyen d'octets que chaque processeur doit échan-

	Arbre 1		Arbre 2		Arbre 3	
Nb de processeurs	moy	ect	moy	ect	moy	ect
4	2499.75	2.68	2499.75	0.43	4036.25	0.43
8	1249.87	1.45	1249.87	0.78	2018.12	0.6
16	624.94	1.25	624.94	0.66	1009.06	0.24
32	312.47	1.6	312.47	1.03	504.53	0.83
64	156.23	1.66	156.23	0.88	252.26	0.71
128	78.12	1.52	78.12	0.75	126.13	0.74
256	39.06	1.59	39.06	0.71	63.06	0.65

FIGURE 5.22 – *Moyenne (moy) et écart type (ect) du nombre d'arêtes obtenu pour chaque processeurs suite au partitionnement.*

ger en une itération de temps et pour un unique *DPMap*, donc une unique quantité de la simulation. Cette table représente également l'écart type maximal obtenus par rapport à cette valeur moyenne. Ainsi, de nouveau, plus l'écart type est proche de la valeur moyenne, plus l'équilibrage des communications effectuées par chaque processeur peut être considéré comme mauvais. On peut observer ici que l'équilibrage de charge en terme de communications est assez mauvais dans ce partitionnement. Cependant l'équilibrage des communications n'est généralement pas considéré dans les problèmes de partitionnements, où l'on cherche plutôt à minimiser le nombre total de communications (pour le partitionnement de graphes) ou le volume total de communications (pour le partitionnement d'hypergraphes). Tout ce que l'on peut conclure de ce résultat est que tous les processeurs n'ont pas la même charge de communications à effectuer et que cette charge peut même aller du simple au double. Cela explique peut être partiellement l'accélération obtenue dans la figure 5.15, sans recouvrement des communications par les calculs.

	Arbre 1		Arbre 2		Arbre 3	
Nb processeurs	moy	ect	moy	ect	moy	ect
4	616.0	168.09	112.0	53.66	148.0	76.84
8	598.0	377.95	98.0	38.31	128.0	42.14
16	437.0	214.77	107.0	42.41	128.0	49.79
32	390.5	238.87	86.5	40.81	135.0	61.18
64	327.0	214.73	95.75	40.02	111.5	42.33
128	266.12	154.3	94.25	31.29	107.5	39.66
256	209.19	113.43	84.25	31.34	94.0	32.52

FIGURE 5.23 – *Moyenne (moy) et écart type (ect) du nombre d'octets à échanger pour chaque processeur, pour chaque* DPMap *et pour une unique itération de temps de la simulation, suite au partitionnement des arbres de la table 5.8.*

Afin de mieux percevoir la charge d'échanges de chaque processeur dans une simulation complète, prenons l'exemple de la simulation artérielle décrite dans la section 5.4, en utilisant les arbres de la table 5.8. Dans la simulation artérielle, il y a onze *DPMap* néces-

sitant des échanges de données, et quatre-vingt mille itérations de temps. La table 5.24 représente alors le nombre moyen total d'octets échangés pour chaque processeur lors de la simulation complète.

Nb processeurs	Arbre 1	Arbre 2	Arbre 3
4	542 Mo	98.6 Mo	130.2 Mo
8	526.2 Mo	86.2 Mo	112.6 Mo
16	384.6 Mo	94.2 Mo	112.6 Mo
32	343.6 Mo	76.1 Mo	118.8 Mo
64	287.8 Mo	84.3 Mo	98.1 Mo
128	234.2 Mo	82.9 Mo	94.6 Mo
256	184.1 Mo	74.1 Mo	82.7 Mo

FIGURE 5.24 – *Moyenne du nombre d'octets total à échanger pour chaque processeur, dans le cadre de la simulation artérielle de la section 5.4, en utilisant les arbres de la table 5.8.*

Les résultats obtenus sont donc relativement bons en terme d'équilibrage de charge. Et les performances obtenues sur *bloodflow-SkelGIS* sont elles aussi relativement bonnes grâce à la mise en place d'un recouvrement des communications par les calculs, comme nous l'avons vu dans la section 5.4.3. Cependant, les performances sont également limitées par les choix qui ont été effectués pour mettre en place un partitionnement. Tout d'abord, notre partitionnement actuel ne s'adresse qu'aux réseaux de type DAG. De plus, nous ne profitons pas de l'efficacité et de l'expertise des partitionneurs existants comme Scotch ou METIS, ni de leur version parallélisées. Enfin, en dupliquant les nœuds, nous ne traitons pas le véritable problème de partitionnement des réseaux et nous ne pouvons garantir son efficacité dans tous les types de simulations et sur tous les types de graphes. En effet, nous avons considéré que les calculs sur les nœuds étaient presque négligeables par rapport aux calculs sur les arêtes, ce qui n'est pas toujours le cas suivant les simulations. De plus, les expériences mesurées dans cette section semblent montrer la nécessité d'améliorer le partitionnement du prototype de SkelGIS. C'est la raison pour laquelle nous avons étudié d'autres méthodes de partitionnement, présentées dans la suite de cette thèse.

5.5.2 Partitionnement avec Mondriaan

Des travaux plus récents sur SkelGIS s'intéressent au véritable problème de partitionnement posé par les réseaux, sans duplication des nœuds. Nous étudions ici la formalisation du problème et deux méthodes de partitionnement. Notons que le but de ce travail est également de retourner vers une définition plus générale des réseaux. Nous prenons en compte ici tout type de réseau, et non plus seulement ceux pouvant être représentés par un graphe dirigé acyclique. En étudiant le véritable problème de partitionnement, nous évitons également la duplication des nœuds sur les processeurs.

5.5.2.1 Formalisation du problème de partitionnement des réseaux

Comme expliqué précédemment, une simulation sur un réseau implique le calcul de deux schémas numériques différents, reliés entre eux par le réseau, créant une certaine dépendance de calcul aux bordures physiques, N_{out}. À partir de deux schémas numériques explicites, il est donc possible d'obtenir un ensemble explicite lui aussi, ou à l'inverse implicite. L'ensemble général obtenu dépend de la simulation en elle-même et ne peut être connu à l'avance. Toutefois, il est possible de définir de façon générale, comme nous l'avons décrit dans la section 5.1, quatre super-étapes dans une simulation sur les réseaux, que nous rappelons ici. Notons $T1$ et $T2$ les deux types d'éléments du réseau qui peuvent être indifféremment associés aux nœuds ou aux arêtes. Une itération t d'une simulation sur un réseau est alors, de façon générale, représentée par les quatre super-étapes suivantes :

1. Communication des éléments T1 à $t-1$
2. Calcul des éléments T2 à t
3. Communication des éléments T2 à $t-1$ ou/et t
4. Calcul des éléments T1 à t

Il est possible de représenter l'ensemble des cas de simulation sur les réseaux en parallèle avec cette définition en quatre super-étapes BSP. Le partitionnement consiste donc en deux problèmes, tout d'abord équilibrer la charge de travail entre les processeurs pour les étapes 2 et 4, et réduire au maximum le volume de communications nécessaire aux étapes 1 et 3. On peut noter que l'étape 2 consiste à calculer $T2$ et a besoin au préalable des communications des éléments $T1$ pour effectuer ce calcul. On parlera alors d'une communication de $T1$ vers $T2$. L'équilibrage de charge consiste donc à équilibrer les deux types d'éléments $T1$ et $T2$, et la réduction du volume de communication consiste à minimiser les communications de $T1$ à $T2$ et de $T2$ à $T1$. Ces deux contraintes doivent être résolues de la même façon quelque soit leur ordre d'apparition et quelque soit donc l'association des types $T1$ et $T2$ aux nœuds et aux arêtes. Pour simplifier la suite de ce travail, nous étudierons le cas précis de la simulation de l'écoulement du sang dans les artères qui a été décrite dans la section 5.4, et dont les quatre super-étapes sont les suivantes :

1. Communication des arêtes à $t-1$
2. Calcul des nœuds à t
3. Communication des nœuds à t
4. Calcul des arêtes à t

La particularité du problème de partitionnement pour les réseaux est donc que des calculs et des communications sont effectués à la fois sur les arêtes et sur les nœuds du graphe. Si les équilibrages de charge des nœuds et des arêtes peuvent être traités de façon distincte, les communications engendrées par leur affectation sont, en revanche, non-distinctes et reliées par le réseau. Tout d'abord, afin de pouvoir traiter à la fois le partitionnement des nœuds et des arêtes en utilisant les outils classiques, qui ne traitent

que les nœuds d'un graphe, nous commençons par changer la représentation du graphe du réseau. Le graphe $G = (V, E)$ avec n nœuds $v_0, \ldots, v_{n-1}$ et m arêtes $e_0, \ldots, e_{m-1}$ est transformé en un nouveau graphe $G' = (V', E')$ où $|V'| = n + m$ et $|E'| = 2m$. Pour chaque arête $e_k = (v_i, v_j) \in E$, un nœud v'_{n+k} est ajouté, et l'arête e_k est coupée en deux arêtes $e'_k = (v_i, v'_{n+k})$ et $e'_{2k} = (v'_{n+k}, v_j)$. Le graphe G' représente donc les arêtes du graphe G comme des nœuds supplémentaires, tout en conservant la connectivité du graphe G. La figure 5.25 illustre un graphe G et le graphe associé G'. Notons que G'

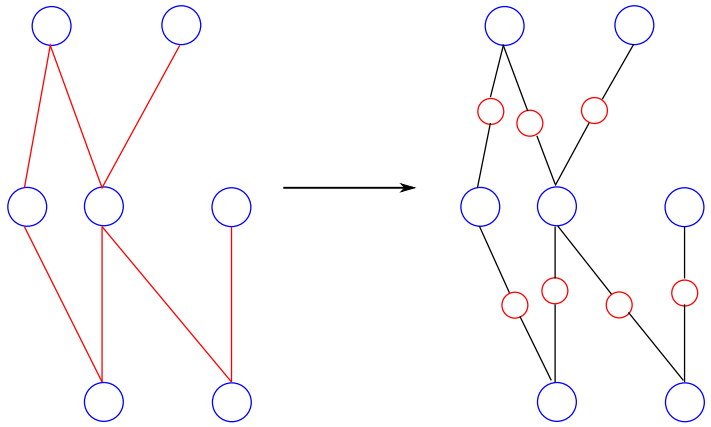
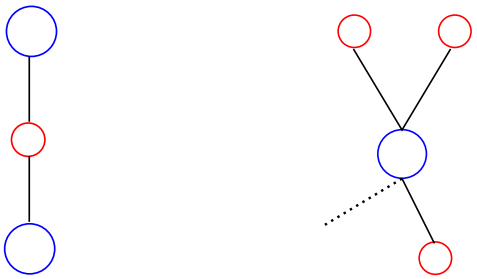


FIGURE 5.25 – *Transformation du graphe G d'un réseau en graphe G'.*

est un graphe biparti avec deux sous-ensembles de nœuds, les rouges et les bleus. Nous n'avons alors dans ce graphe que des liens du type bleu-rouge mais pas de liens du type bleu-bleu ou rouge-rouge. Dans ce nouveau graphe G', les étapes de communications 1 et 3, représentées dans les figures 5.26(a) et 5.26(b), deviennent :

— Communication des nœuds rouges vers les nœuds bleus
— Communication des nœuds bleus vers les nœuds rouges



(a) Étape de communication 1, des nœuds rouges vers les nœuds bleus de G'.

(b) Étape de communication 3, des nœuds bleus vers les nœuds rouges de G'.

FIGURE 5.26 – *Étapes de communication 1 et 3.*

Dans la suite, les nœuds bleus représenteront donc les nœuds du réseau initial G, et les nœuds rouges les arêtes du réseau initial G. Étant donné le graphe biparti G' ainsi défini, l'ensemble des nœuds V' du graphe est composé de deux parties V^r et V^b, telles que $V' = V^r \cup V^b$ et $V^r \cap V^b = \emptyset$. Le problème de partitionnement du graphe G' consiste alors à partitionner V^r et V^b en p parties telles que $V^r = \bigcup_i V_i^r$ et $V^b = \bigcup_i V_i^b$ et telles que pour $i \neq j \in [\![0, p[\![, V_i^r \cap V_j^r = \emptyset$ et $V_i^b \cap V_j^b = \emptyset$. Le partitionnement de G' ainsi défini doit minimiser le volume de communications entre les processeurs, tout en répondant aux deux contraintes d'équilibrage de charge suivantes :

$$\omega(V_i^r) \leq (1+\epsilon)\frac{\omega(V^r)}{p} \tag{5.9}$$

$$\omega(V_i^b) \leq (1+\epsilon)\frac{\omega(V^b)}{p} \tag{5.10}$$

où $\omega(A)$ représente le poids total des nœuds d'un ensemble $A \in V$, et où ϵ représente le pourcentage de tolérance dans l'équilibrage de charge.

Pour résoudre ce problème de partitionnement, le modèle de partitionnement d'hypergraphe (défini dans la section 2.3 de cette thèse) est utilisé sur le graphe G'. Deux méthodes de partitionnement différentes sont étudiées et sont détaillées dans les deux sections suivantes.

5.5.2.2 Méthode à partitionnement unique

La première méthode qui a été étudiée, appelée la méthode à partitionnement unique, est composée de deux étapes que nous allons expliquer en détails dans cette partie :

1. L'étape de communication des nœuds bleus vers les nœuds rouges est, tout d'abord, transformée en un problème de partitionnement d'hypergraphe, qui permet de distribuer les nœuds rouges sur les différents processeurs.
2. Une heuristique est ensuite appliquée pour distribuer les nœuds bleus sur les processeurs, tout en prenant en compte la distribution des nœuds rouges qui a été faite au préalable.

Afin d'effectuer la première étape de cette méthode, une matrice A de taille $m \times n$ est créée et a pour but de représenter les communications des nœuds bleus vers les nœuds rouges. Les lignes de la matrice A représentent les nœuds bleus du graphe G', et les colonnes de la matrice A représentent, quant à elles, les nœuds rouges du graphe G'. Si une communication est nécessaire d'un nœud bleu vers un nœud rouge, une valeur 1 est placée dans A aux coordonnées correspondantes. Si l'on distribue les colonnes de la matrice A (les nœuds rouges) aux processeurs, le volume de communication sera minimisé si et seulement si les éléments non nuls de chaque ligne sont répartis sur le minimum de processeurs différents. La distribution des colonnes de la matrice A revient en fait à procéder à un partitionnement d'hypergraphe suivant une dimension. Il s'agit donc du partitionnement *row-net model* [29], décrit dans l'état de l'art de cette thèse dans la section 2.3, qui consiste à distribuer les colonnes de la matrice A (les nœuds de

l'hypergraphe), tout en cherchant à minimiser le nombre de coupures sur une ligne de A (une hyper-arête). La figure 5.27 donne un exemple de réseau, la matrice A qui lui est associée, et enfin l'hypergraphe $H_r(A)$ qui lui est également associé.

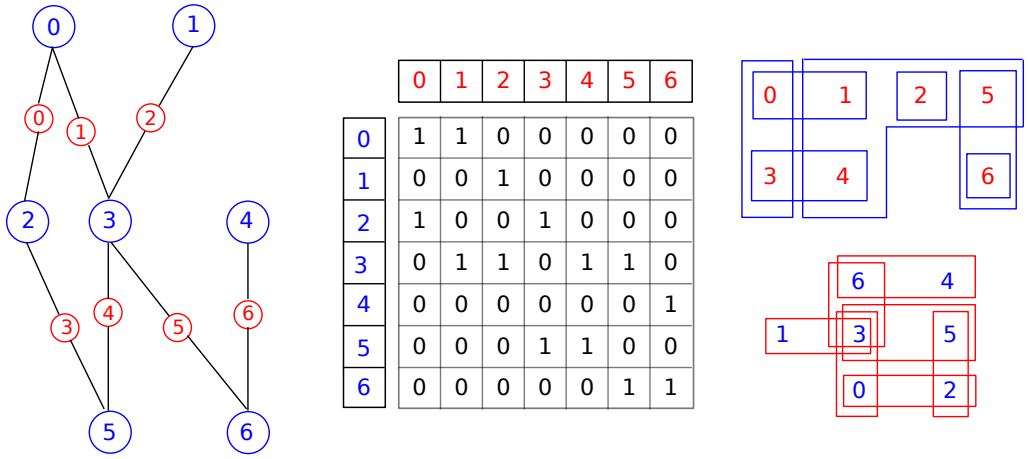


FIGURE 5.27 – *Exemple de réseau G' avec la matrice A et les hypergraphes H_r et H_c qui y sont associés*

Le partitionnement de l'hypergraphe H_r associé à un réseau initial G peut être effectué avec le partitionneur Mondriaan [120] dans un mode à une dimension. Les nœuds rouges de G', c'est-à-dire les arêtes du réseau, sont alors distribués en respectant la contrainte (5.9) et en cherchant à minimiser les communications des nœuds bleus vers les nœuds rouges.

Une fois cette première étape effectuée, le partitionnement de V^r est terminé et respecte la contrainte d'équilibrage (5.9). Les nœuds rouges et bleus de G' étant connectés par le réseau, il est important de trouver une heuristique permettant de distribuer les nœuds bleus en tenant compte de la distribution des nœuds rouges, afin de minimiser le nombre de communications nécessaires. Le problème de partitionnement pour les nœuds bleus est illustré dans la figure 5.28. En effet, étant donné quatre nœuds rouges déjà distribués aux processeurs p_0, p_1 et p_2, comment choisir parmi ces trois processeurs celui qui sera attribué au nœud bleu.

Afin d'expliquer l'heuristique mise en place pour le partitionnement des nœuds bleus, quelques définitions sont nécessaires. Pour un nœud $v \in V^b$, $deg(v)$ représente le nombre de nœuds rouges adjacents à v dans G', ou en d'autres termes le nombre d'arêtes adjacentes à v dans G. $deg(v, P_i)$ représente le nombre de nœuds rouges adjacents à v et qui ont été distribués au processeur P_i. L'heuristique choisit d'attribuer le nœud v au processeur P_i de façon à maximiser $deg(v, P_i)$, et de façon à vérifier la contrainte d'équilibrage (5.10). Notons que si il existe $i, j \in [\![0, p[\![$ qui satisfont $deg(v, P_i) = deg(v, P_j)$, alors $P(v) = P_i$ si et seulement si $\omega(V_i^b) < \omega(V_j^b)$, sinon $P(v) = P_j$. Grâce à cette heuristique les nœuds bleus de G' sont partitionnés en tentant de respecter la contrainte d'équilibrage de charge (5.10), tout en minimisant les communications des nœuds rouges

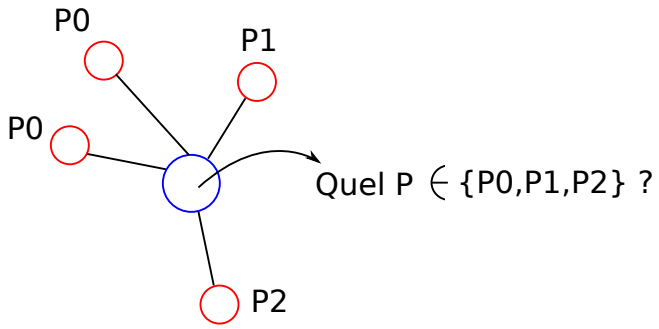


FIGURE 5.28 – *Problème de partitionnement pour les nœuds bleus de G'.*

vers les nœuds bleus. En effet, en reprenant l'exemple de la figure 5.28, l'heuristique décrite choisie idéalement d'assigner le nœud au processeur P_0. Dans ce cas seules deux communications des nœuds rouges vers les nœuds bleus sont nécessaires. À l'inverse, si un autre processeur avait été choisi, trois communications des nœuds rouges vers les nœuds bleus auraient été nécessaires.

5.5.2.3 Méthode à partitionnement double

La deuxième méthode de partitionnement qui a été étudiée, appelée la méthode à partitionnement double, est elle composée des trois étapes suivantes :

1. Tout comme dans la première méthode, l'étape de communication des nœuds bleus vers les nœuds rouges est transformée en un problème de partitionnement d'hypergraphe, qui permet de distribuer les nœuds rouges sur les différents processeurs.
2. L'étape de communication des nœuds rouges vers les nœuds bleus est ensuite transformée en un problème de partitionnement d'hypergraphe, qui permet de distribuer les nœuds bleus à leur tour.
3. Une dernière étape de permutation est mise en place pour permettre de réaffecter certains processeurs à certaines partitions et ainsi d'améliorer la minimisation du nombre de communications.

La première étape de cette méthode est exactement la même que la première étape de la méthode à partitionnement unique décrite précédemment. Les communications des nœuds bleus vers les nœuds rouges y sont exprimées dans une matrice creuse A correspondant à la matrice d'incidence du graphe G. Puis la matrice A est transposée en un hypergraphe H_r, où les lignes de A représentent les hyper-arêtes et les colonnes les nœuds de H_r. Un partitionnement *row-net* à une dimension est effectué sur l'hypergraphe H_r avec le partitionneur Mondriaan [120].

La deuxième étape de cette méthode est, quant à elle, la transposée de la première étape. Les communications des nœuds rouges vers les nœuds bleus peuvent être représentées par la matrice A^T, transposée de la matrice A. Cette matrice pourrait à son tour être transformée en un hypergraphe $H_r(A^T)$ sur lequel serait appliqué un partitionnement à

une dimension de type *row net*. Toutefois, il est plus simple de conserver la matrice A mais d'y appliquer un partitionnement à une dimension de type *column net* sur un hypergraphe noté $H_c(A) = H_r(A^T)$ qui va, de la même façon, partitionner les nœuds bleus et réduire les communications des nœuds rouges vers les nœuds bleus. La figure 5.27 illustre ces deux premières étapes du partitionnement et les deux hypergraphes $H_r(A)$ et $H_c(A)$.

La différence principale entre la première et la deuxième étape de cette méthode est le nombre de nœuds présents dans chaque hyper-arête. En effet, dans la première étape, le nombre de nœuds dans chaque hyper-arête est égal au degré du nœud bleu correspondant. Dans la deuxième étape, en revanche, le nombre de nœuds dans chaque hyper-arête est égal à deux puisque chaque nœud rouge de G' est uniquement relié à deux nœuds bleus. Par conséquent, le partitionnement d'hypergraphe row-net (column-net) appliqué dans la deuxième étape de la méthode est équivalent à un partitionnement de graphe standard où le nombre d'arêtes coupées doit être minimisé. Un partitionneur de graphe pourrait donc être utilisé pour résoudre la deuxième étape de la méthode, comme par exemple METIS [77] ou Scotch [100]. Toutefois, il est également possible d'utiliser le même partitionneur d'hypergraphes Mondriaan [120], et ainsi d'en abuser légèrement [53] tout en payant le coût supplémentaire en temps CPU. En effet, le partitionnement d'un hypergraphe n'est qu'une généralisation du partitionnement de graphe, cette méthode peut donc être utilisée pour le partitionnement de graphe mais demandera plus de calculs du fait de la généralisation qui y est appliquée.

Une fois la première et la deuxième étape effectuées, les nœuds rouges et les nœuds bleus du graphe G' sont attribués parmi les processeurs disponibles en respectant les deux contraintes d'équilibrage (5.9) et (5.10). Toutefois les deux partitionnements qui ont été effectués sont totalement indépendants et ne tiennent donc pas compte des connections qui relient les nœuds bleus et les nœuds rouges dans le réseau.

La troisième étape va servir à prendre en compte ces liaisons et ainsi à réduire les communications engendrées par les deux partitionnements distincts. On obtient donc deux partitionnements, et l'on souhaite pouvoir réaffecter les processeurs dans ces distributions afin d'améliorer le volume de communications. Il s'agit donc d'un problème d'affectation dans un graphe biparti. Ce problème est équivalent à effectuer une permutation des processeurs de façon optimale. Tout d'abord, nous définissons une matrice W de taille $p \times p$ (p étant le nombre de processeurs) pour exprimer le nombre de communications évitées si une permutation était effectuée entre deux processeurs donnés. Le calcul de cette matrice W est basée sur la matrice A, puisque celle-ci représente les communications des nœuds bleus vers les nœuds rouges, et à l'inverse, en utilisant sa transposée, des nœuds rouges vers les nœuds bleus.

Dans la matrice A, $a_{i,j} = 1$ si un nœud bleu i est relié à un nœud rouge j. Une fois le partitionnement des nœuds bleus et rouges effectués, $\Phi(i)$ représente le processeur attribué pour le nœud bleu i, et $\Psi(j)$ représente le processeur attribué pour le nœud rouge j. La matrice W est alors définie comme suit

$$w_{s,t} = \sum_{i \,:\, \Phi(i)=s} \delta_i(t) \tag{5.11}$$

$$\delta_i(t) = \begin{cases} 1 & \text{si } \exists j \ : \ a_{i,j} = 1 \wedge \Psi(j) = t \\ 0, & \text{sinon.} \end{cases} \tag{5.12}$$

La taille de la matrice W dépend du nombre de processeurs utilisés et la matrice est généralement dense. L'élément $w_{s,t}$ de la matrice représente le nombre de communications évitées, des nœuds bleus vers les nœuds rouges, si le processeur s est permuté avec le processeur t. La matrice W peut donc être identifiée comme un graphe biparti complet G_w, illustré dans la figure 5.29. Un moyen de calculer la meilleure permutation possible de processeurs est alors de calculer le *couplage* ou l'*appariement* (ou le *matching* en anglais) maximum du graphe biparti complet G_w.

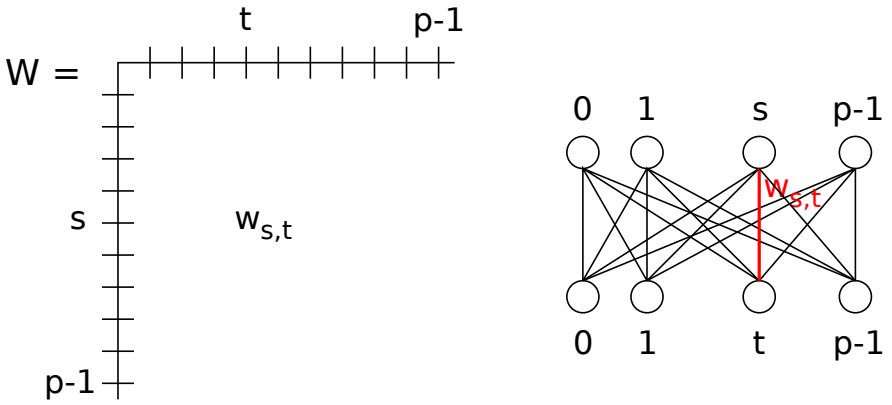


FIGURE 5.29 – *La matrice W et le graphe biparti complet auquel la matrice peut être identifiée G_w.*

Le couplage maximum de G_w est équivalent à un problème d'assignement qui peut être résolu en $O(p^4)$ en utilisant l'*algorithme Hongrois*, publié par Harold Kuhn en 1955 [83] ; et qui peut également être résolu en $O(p^3)$ en utilisant une amélioration de cet algorithme. Si le nombre de processeurs p est grand, des approximations linéaires de cet algorithme peuvent également être utilisées.

La matrice W représente les communications évitées, mais uniquement pour les communications des nœuds bleus vers les rouges. Le problème similaire peut être résolu pour la matrice A^T et permettra de réduire les communications des nœuds rouges vers les nœuds bleus également. Une matrice W' est alors calculée n'est pas égale à W^T. Toutefois, en calculant de la même façon que W la matrice W', en utilisant A^T au lieu de A, la valeur $w'(s,t)$ de W' représente le nombre de communications évitées en échangeant le processeur t avec le processeur s. Afin de maximiser les communications évitées par permutation, l'algorithme Hongrois doit alors être appliqué sur $\overline{W} = W + (W')^T$.

Cette deuxième méthode respecte donc les deux contraintes d'équilibrage (5.9) et (5.10) en appliquant deux partitionnements d'hypergraphe à une dimension sur la matrice A. Par la suite, afin de lier les deux distributions obtenues et d'en minimiser le volume des communications obtenu, un couplage maximum est appliqué sur la matrice $\overline{W}$.

5.5.2.4 Résultats

A l'heure actuelle aucun résultat n'a encore été établi pour ces deux nouvelles méthodes de partitionnement des réseaux. Ces nouvelles méthodes s'inscrivent dans un cadre plus général que l'implémentation actuelle de SkelGIS, puisqu'elles généralisent le problème aux graphes et non plus seulement aux DAG. Un certain nombre d'adaptations est en cours pour permettre l'utilisation de ces méthodes de partitionnement, puis pour les évaluer sur le cas de test présenté dans la section 5.4. Une intuition sur les résultats attendus est toutefois possible. Il semble, à première vue, que la méthode à partitionnement simple soit plus intéressante. En effet, elle répond strictement aux deux contraintes d'équilibrage de charges des équations (5.9) et (5.10), tout comme la méthode à double partitionnement. Toutefois, elle distribue directement les sommets bleus en tenant compte des sommets rouges, alors que la méthode à double partitionnement effectue deux partitionnements totalement indépendants avant d'essayer de les rapprocher par un matching. Cela laisse penser que malgré le matching effectué la méthode à partitionnement double ne pourra être aussi efficace, en terme de volume de communications, que dans la première méthode. Mais essayons de décrire un modèle de coût pour prévoir les résultats de ces méthodes de façon plus formelle.

Étant donné une simulation sur les réseaux implémentée en SkelGIS, nous pouvons noter T^i_{comp} le temps passé dans les calculs pour le processeur i. Étant donné $T^i_{1_{comp}}$ et $T^i_{2_{comp}}$, le temps écoulé dans le calcul des nœuds et des arêtes du réseau (ou vice versa), pour le processeur i, nous considérons que $T^i_{comp} = T^i_{1_{comp}} + T^i_{2_{comp}}$. Les deux méthodes présentées dans cette section respectent strictement les deux contraintes d'équilibrage de charges (5.9) et (5.10). Cela signifie que pour tout processeurs i et j, tel que $i \neq j$, $T^i_{1_{comp}} \approx T^j_{1_{comp}}$ et $T^i_{2_{comp}} \approx T^j_{2_{comp}}$. On peut alors considérer que le temps total de calcul de la simulation, T_{comp}, est égale à $\max\limits_{0 \leq i < p} T^i_{comp}$ mais aussi que quelque soit le processeur i du programme, $T_{comp} \approx T^i_{comp}$. En procédant de même pour les communications, avec $T^i_{comm} = T^i_{RB_{comm}} + T^i_{BR_{comm}}$ le temps passé dans les communications pour le processeur i, où $T^i_{RB_{comm}}$ et $T^i_{BR_{comm}}$ sont les temps respectifs passés dans les communications des sommets rouges vers les sommets bleus et des sommets bleus vers les sommets rouges. Nous obtenons alors le temps total dépensé en communication, pour la simulation, $T_{comm} = \max\limits_{0 \leq i < p} T^i_{comm}$. Étant donné S, le nombre d'étapes de calcul dans la simulation, en considérant que le temps de synchronisation des processeurs est inclus dans T_{comm}, nous nous retrouvons alors avec un temps total d'exécution du programme T tel que

$$T \approx S \times (T_{comp} + T_{comm}). \tag{5.13}$$

Cette analyse, proche du modèle de coût BSP, est possible du fait que les calculs effectués dans une simulation sur les réseaux se rapportent très facilement au modèle BSP, comme nous l'avons vu. Nous pouvons alors analyser plus en détails les différents résultats attendus pour les deux méthodes de partitionnement proposées. Notons que dans l'analyse qui va suivre, il est considéré que l'optimisation de recouvrement des

communications par les calculs est mise en place systématiquement. Cela signifie qu'il est possible d'effectuer des communications pendant les calculs.

Si $T_{comm} \ll T_{comp}$, alors le temps passé dans les calculs est beaucoup plus important que le temps passé dans les communications. Étant donné que les deux méthodes de partitionnement respectent strictement les deux contraintes d'équilibrage de charges des équations (5.9) et (5.10), aucune méthode ne peut théoriquement être mise en avant par rapport à l'autre.

Si $T_{comm} \gg T_{comp}$, à l'inverse, le temps passé dans les communications est beaucoup plus important que le temps de calcul. Dans ce cas, un recouvrement des communications par les calculs ne suffira plus à combler un volume de communication trop important, et il est probable que la méthode à partitionnement simple obtienne de meilleures performances.

Si $T_{comm} \approx T_{comp}$, il est alors difficile de déterminer si l'une des deux méthodes aura de meilleures performances.

5.6 Conclusion

Nous avons abordé dans ce chapitre l'application du modèle SIPSim sur un cas particulier de simulation scientifiques, les simulations sur les réseaux. Ce type de simulation a la particularité de pouvoir appliquer deux maillages différents et des schémas numériques différents sur les nœuds et sur les arêtes d'un graphe. En d'autres termes ce type de simulation représente une composition de maillage, effectuée sur une structure de données totalement irrégulière, un réseau. Un réseau n'étant pas un maillage, la difficulté de ce travail résidait tout d'abord dans le fait de pouvoir appliquer le modèle SIPSim à des notions plus évoluées de simulation. Nous avons donc tout d'abord étudié les réseaux en détails, et en quoi ce type de simulation pouvait s'appliquer au modèle SIPSim. Nous avons donc montré la souplesse du modèle SIPSim et la possibilité de son implémentation efficace sur des cas de simulation complexes.

Un réseau étant une structure de données irrégulière, et donc complexe, nous avons fait le choix de réduire le problème à des réseaux de type DAG, et à des maillages à une dimension. Malgré cette simplification, l'implémentation de SkelGIS pour les réseaux a soulevé des points intéressants de partitionnement de graphe et de structure de données irrégulière efficace. Les résultats obtenus ont montré l'efficacité de SkelGIS sur des architectures à mémoire partagée et distribuée. Une fois encore, à partir d'un code séquentiel donné, nous avons comparé la parallélisation SkelGIS avec une autre implémentation parallèle, cette fois en OpenMP. Les comparaisons ont été faites en terme de performances, mais aussi, et surtout, en terme d'effort de programmation. La programmation d'une structure de données irrégulière, telle qu'un réseau, est une tâche difficile et longue. SkelGIS cache non seulement l'implémentation de cette structure de données complexe, mais

également son optimisation et sa distribution. Dans ce cas alors, la simplicité d'utilisation de SkelGIS, et plus généralement des solutions SIPSim, prend tout son sens et montre un effort de programmation très faible.

Pour terminer, nous avons détaillé des travaux en cours de réalisation visant à obtenir un meilleur partitionnement des réseaux sur les processeurs, et donc probablement de meilleures performances. Ces travaux s'inscrivent également dans une généralisation du problème de simulations sur les réseaux, puisque les partitionnements étudiés se généralisent aux graphes et non plus seulement aux DAG.

Conclusion 6

Sommaire

6.1 Bilan

Les travaux présentés dans cette thèse se sont découpés en trois parties principales. Après un état de l'art détaillé sur l'ensemble de notions nécessaires à la compréhension de cette thèse, nous avons présenté le modèle de programmation parallèle implicite SIPSim. Ce modèle, qui propose une approche systématique permettant d'implémenter des solutions de parallélisme implicite pour les simulations scientifiques, s'inspire des trois grandes familles de solutions de parallélisme implicites : les bibliothèques de parallélisme implicite générales, et notamment STAPL [25], les solutions à patrons et les bibliothèques de squelettes algorithmiques, Pregel [88] par exemple, et enfin les langages, bibliothèques ou frameworks spécifiquement implémentés pour un domaine précis du calcul scientifique, comme par exemple les langages OP2 [60, 94] et Liszt [50]. Le modèle SIPSim recommande l'implémentation de quatre composants pour obtenir une solution de parallélisme implicite à la fois souple et donc relativement générale, mais également suffisamment spécifique pour proposer de bonnes performances. Ces quatre composants sont (1) une structure de données distribuée, qui peut représenter un maillage ou une structure de composition de maillages, (2) un composant permettant d'appliquer des données sur la structure de données distribuée (ces données sont elles-mêmes distribuées), (3) une notion d'applicateur permettant de cacher les communications entre les processeurs à l'utilisateur et d'appliquer des opérations séquentielles implémentées par l'utilisateur, et enfin (4) des interfaces de programmation, et notamment le concept très important d'itérateur à l'origine de la souplesse du modèle.

Nous avons ensuite présenté une première implémentation du modèle de programmation parallèle implicite SIPSim, sous la forme d'une bibliothèque C++ de type header-only, appelée SkelGIS. Cette première implémentation visait à valider le modèle sur les maillages cartésiens à deux dimensions, très utilisés dans les simulations numériques. Les quatre composants de l'approche SIPSim ont été implémentés pour ce type de maillages, et certaines optimisations y ont été appliquées, notamment la technique de spécialisation partielle de template, afin de fournir une solution compétitive en terme d'efficacité. Nous avons illustré deux cas d'applications de cette première implémentation afin de pouvoir l'évaluer à la fois en terme de performances, mais aussi en terme d'effort de programmation. Le premier cas d'application a été l'implémentation de la résolution de l'équation de la chaleur par la méthode numérique des différences finies. Le deuxième cas réel d'application fût, quant à lui, la résolution des équations de Saint Venant par une méthode des volumes finis, décrite avec précision dans les travaux de Bouchut [20] et dans la thèse de Delestre [48]. Les mêmes méthodes d'évaluation ont été appliquées pour ces deux cas d'applications. À partir d'un même code séquentiel, sans y ajouter d'optimisation particulière, deux versions parallèles ont été produites. La première en MPI a permis d'obtenir, avec le minimum d'effort de programmation possible (structure de données distribuée simple, utilisation de topologies et types dérivés MPI, utilisation des communications MPI), une version parallèle obtenant une bonne scalabilité proche de l'accélération idéale. La deuxième a été écrite en utilisant la bibliothèque SkelGIS. Les deux versions parallèles ont ainsi pu être comparées en terme d'effort de programmation

et en terme de performances. Pour l'équation de la chaleur, comme pour les équations de Saint Venant, l'effort de programmation demandé par SkelGIS est beaucoup moins important que celui demandé par la version MPI. Les temps d'exécution obtenus sont meilleurs pour les versions SkelGIS, mais les accélérations sont, quant à elles, légèrement supérieures pour les versions MPI, qui proposent une meilleure scalabilité sur plus de processeurs. Nous avons pu noter, sur ces deux cas d'applications, que les surcoûts introduits par le parallélisme implicite de SkelGIS diminuaient en augmentant la taille du maillage à traiter. Nous avons donc montré, dans ces premiers travaux, que le modèle SIPSim pouvait être implémenté de façon efficace et qu'il produisait des codes demandant un effort de programmation très faible face à une implémentation MPI.

La suite du manuscrit a ensuite présenté le cas des compositions de maillages typiques aux simulations sur les réseaux. Nous avons commencé par introduire clairement la notion de réseau, afin d'étudier pourquoi le modèle SIPSim pouvait s'appliquer à ce type de simulations. Nous avons ensuite décrit l'implémentation des quatre composants SIPSim dans la bibliothèque SkelGIS pour les réseaux. Nous avons plus particulièrement décrit l'implémentation de la structure de données distribuée pour les réseaux, dont l'optimisation fût très importante pour obtenir de bonnes performances. Cette implémentation propose une version parallèle et ré-indexée du format CSR (Compressed Sparse Row), et permet notamment d'accéder aux voisinages des éléments du réseau en $O(1)$. Un cas d'application réel a ensuite été détaillé, il s'agit d'une simulation d'écoulement du sang dans le réseau artériel, dont les détails sont donnés dans les travaux de Wang et Al [126]. Tout comme pour l'évaluation de SkelGIS sur maillages cartésiens, nous avons évalué cette simulation en terme d'effort de programmation et en terme de performances. À partir d'une même version séquentielle, de nouveau, et sans optimisation particulière de cette version séquentielle, deux versions parallèles ont été implémentées. La première parallélisation, de type OpenMP coarse-grain, requiert un effort de programmation supérieur pour un temps d'exécution et une accélération moins intéressante que la version SkelGIS. Nous avons ensuite étudié la scalabilité de la version SkelGIS sur des clusters de taille importante en montant jusqu'à plus de huit milles cœurs de calcul. Une fois encore, les surcoûts introduits par le parallélisme implicite de SkelGIS peuvent diminuer en augmentant la taille de réseau à traiter. Cette implémentation de la méthode SIPSim a montré encore davantage l'intérêt de ce modèle pour des cas d'applications complexes. De plus, cette implémentation a montré la souplesse du modèle SIPSim et son adaptativité. Ce modèle ne se cantonne pas aux simulations simples mais à toutes sortes de simulations. Enfin, nous avons évoqué les premiers travaux menés en collaboration avec Rob Bisseling sur le partitionnement de réseau. Une nouvelle implémentation de SkelGIS est en cours d'élaboration pour pouvoir évaluer ces nouvelles méthodes de partitionnement.

6.2 Perspectives

Les travaux présentés dans cette thèse ouvrent de nombreuses perspectives de recherche que nous allons essayer de détailler.

Tout d'abord, comme nous l'avons déjà évoqué, les travaux OP2 et Liszt nous montrent, par leurs concepts relativement proches de la méthode SIPSim, qu'il est possible d'implémenter la méthode SIPSim pour les maillages non structurés et les méthodes des éléments finis. Un prototype d'implémentation de SkelGIS pour ce type de maillages serait un premier point intéressant pour valider encore davantage le modèle SIPSim et pour pouvoir comparer les performances obtenues pour les trois solutions. La topologie d'un maillage non-structuré est un graphe, toutefois les points et les arêtes d'un maillage non-structuré forment des géométries identiques. Par exemple, un maillage non-structuré est souvent composé de triangles en deux dimensions, ou de tétraèdres en trois dimensions. Il s'agit donc d'une structure plus irrégulière qu'un maillage cartésien mais plus régulière qu'un réseau. Comme nous l'avons expliqué dans la section 2.3 de l'état de l'art, le partitionnement des maillages non-structurés est un problème qui a largement été traité et qui ne posera pas de problèmes majeurs. La difficulté de ce travail résidera donc principalement dans l'étude des résolutions sur maillages non-structurés afin de proposer des interfaces de programmation (itérateurs et accès aux voisinages) adaptées aux simulations.

Le travail effectué sur les réseaux et présenté dans cette thèse est un premier pas vers une généralisation du problème de composition de maillages, qui sera une contribution intéressante en parallélisme implicite. Les travaux présentés dans cette thèse se sont limités au cas des réseaux de type DAG et au cas de composition de maillages à une dimension. Une généralisation vers des réseaux quelconques, et donc des graphes généraux (orientés ou non, avec ou sans cycles) sera tout d'abord intéressante. Cette adaptation a déjà été initiée grâce aux travaux effectués sur le partitionnement de réseau, qui généralisent le problème de partitionnement aux graphes. Une généralisation à tous types de maillages dans les nœuds et les arêtes d'un réseau, est en revanche un travail qui ouvrent beaucoup plus de perspectives et aussi de difficultés, et une amélioration du modèle SIPSim pourrait être envisagée en conséquence. En effet, il parait évident suite au travail effectué sur les réseaux, que pour proposer de la composition de maillages quelconque il faut se diriger vers une composition de DDS du modèle SIPSim. Par exemple, il sera possible de définir que les nœuds et les arêtes d'un réseau contiennent des maillages cartésiens ou des maillages non structurés, eux-mêmes implémentés dans la solution. Cette évolution va se traduire par un étude intéressante des notions de voisinages imbriqués. Afin de rendre les concepts plus simples à implémenter, le modèle SIPSim devra alors probablement évoluer vers des recommandations plus strictes sur l'implémentation des DDS. On peut par exemple imaginer une implémentation des DDS plus proche de la programmation par composants [14,15] où les branchements entre les DDS se feront par le voisinage. Une généralisation à tout type de compositions de maillages sera alors également envisageable, comme par exemple les maillages hybrides qui permettent de relier des maillages entre eux par l'intermédiaire d'un maillage non-structuré.

À plus long terme, il pourrait également être intéressant de faire évoluer le modèle SIPSim pour gérer d'autres types de maillages, mais pour lesquels les concepts introduits s'appliqueraient toujours. Tout d'abord, nous pouvons évoquer les méthodes numériques qui utilisent des maillages adaptatifs. Dans ce cas, à chaque itération de temps, un raffinement du maillage est calculé, et peut potentiellement modifier le maillage de l'itération précédente, pour obtenir plus de précision à certains endroits du domaine étudié. Ce type de maillages introduira deux difficultés principales. Tout d'abord il faudra envisager dans le modèle une re-distribution du maillage à chaque itération de temps, afin de conserver un bon équilibrage de charge. Il existe des méthodes de partitionnement permettant de redistribuer de façon peu coûteuse un maillage adaptatif [85, 121, 124]. La deuxième difficulté sera de permettre la modification d'un maillage et de sa topologie pendant l'exécution. Nous pouvons ensuite évoquer les méthodes numériques utilisant des multigrilles. Ce terme peut faire naturellement penser aux compositions de maillages, mais est en fait un concept très différent. Ces méthodes visent, en effet, à faire converger la solution d'une simulation plus rapidement, par l'utilisation d'une hiérarchie de grilles ou de maillages. Un même calcul est effectué sur un maillage fin, et sur le même maillage rendu plus grossier, pour permettre, par un système proche de l'interpolation, une correction globale du résultat. Ce type de résolution est, a priori, envisageable avec le modèle SIPSim par l'utilisation de plusieurs DDS à la fois. Toutefois, les techniques permettant, à partir d'un maillage fin, d'obtenir un maillage grossier peuvent impliquer, là encore, une modification, ou une création de DDS à l'exécution de la simulation.

Nous avons évoqué les perspectives possibles de ce travail en termes de types de maillages. Une autre évolution possible du travail présenté dans cette thèse concerne également le type de schémas numériques étudiés. Dans le travail effectué sur les maillages cartésiens, nous avons, en effet, limité notre analyse aux schémas numériques explicites. Dans notre implémentation de SkelGIS pour les réseaux, en revanche, nous avons évoqué les schémas numériques implicites, alors presque inévitables. Il semble en fait que le type de programmation mis en place dans le modèle SIPSim rendent possible la résolution de schémas numériques implicites. Rappelons qu'un schéma explicite calcule les nouvelles valeurs de l'itération $t+1$ uniquement en utilisant les résultats de l'itération t, alors qu'un schéma numérique implicite se sert également des résultats de l'itération courante $t+1$ pour effectuer ses calculs. En d'autres termes, un schéma numérique implicite introduit un problèmes d'ordonnancement dans les calculs à effectuer dans une itération. On peut envisager deux cas d'ordonnancement :

— Une quantité doit être calculée avant une autre : c'est le cas qui parait le plus probable dans une simulation. La dépendance en temps vient alors du fait qu'une quantité a besoin d'une autre pour être calculée. Le modèle SIPSim est d'ores et déjà capable de traiter ce type de simulations, comme nous l'avons évoqué dans la section 5.2. En effet, l'appel à un applicateur effectuant tout d'abord les communications nécessaires, puis après seulement l'appel à l'opération utilisateur, plusieurs appels d'applicateurs dans une itération de temps rendent possible le calcul de schémas numériques implicites. Toutefois, une clarification du modèle et

de son utilisation dans les différents cas de schémas est à établir pour faciliter sa compréhension.

— Le calcul d'une quantité est ordonnancée : cela signifie alors que les calculs sur le maillage sont ordonnancés pour une même quantité. Le modèle SIPSim ne sait pas traiter ce cas d'ordonnancement puisque le concept d'itérateur part justement du principe que les éléments du maillages (ou nœuds actifs, dans le TAO du parallélisme [103]) ne sont pas ordonnés.

Enfin, la dernière perspective que nous souhaitons présenter dans ce travail de thèse, est la mise en place d'une solution de parallélisme implicite basée sur le modèle SIPSim, pouvant être mise en production. Il s'agirait donc hypothétiquement d'un outil de parallélisme implicite permettant de définir des simulations basées sur des maillages cartésiens à n dimensions, des maillages non-structurés, des maillages adaptatifs, des maillages composés ou encore des maillages multi-grilles. Pour cela, il serait probablement intéressant d'envisager une nouvelle implémentation du modèle SIPSim, autre que la bibliothèque SkelGIS. Notre intérêt se porte particulièrement sur la proposition d'un langage de domaine spécifique simple d'utilisation, mais proche d'un programme séquentiel pour ne pas désorienter les habitudes des utilisateurs. La compilation du langage, que nous appellerons pré-compilation, vers une application C++ haute performance serait alors effectuée, avant que celle-ci ne puisse à son tour être compilée par un compilateur natif du C++. La pré-compilation, ou encore appelée la compilation source-à-source, permet la mise en place, de façon relativement libre, d'un certain nombre d'optimisations qu'il est difficile d'appliquer dans une bibliothèque header-only telle que SkelGIS. En effet, dans une bibliothèque C++, rendre certaines optimisations de code implicite, tout en proposant à l'utilisateur de coder ses propres opérations, implique l'utilisation de mécanismes très avancés, tels que la méta-programmation et la mise en place d'expression templates [3], qui rendent rapidement un code difficile à entretenir et surtout à faire évoluer. Si ces mécanismes ont prouvé leur efficacité, une solution par compilation semble plus durable. Nous envisageons, par exemple, d'utiliser une sous-partie du langage Python comme langage de description pour l'utilisateur, ce qui permettrait d'hériter de la simplicité de ce langage, très facile à apprendre. L'implémentation d'un nouveau compilateur pour cette sous-partie du langage Python transformerait alors le code utilisateur en un code C++ parallèle et optimisé. Cette solution se porterait d'ailleurs plus facilement à la proposition de parallélisme implicite sur plusieurs types d'architectures parallèles (mémoire partagée CPU et GPU et architectures hybrides).

Notons enfin que l'étude d'un modèle de prédiction de performances pourrait accompagner ce nouveau langage. En effet, comme nous l'avons évoqué dans cette thèse à plusieurs reprises, le modèle SIPSim peut s'apparenter au modèle BSP. L'appel à un applicateur peut être identifié à un ensemble de super-étapes de communications et de calculs. Il paraît donc possible d'établir un modèle de coût en héritant de celui proposé par BSP, et présenté dans la section 2.1.2 de l'état de l'art. Toutefois, certaines optimisations pourraient limiter la fiabilité de ce modèle, notamment les optimisations de recouvrement des communications par le calcul, mais aussi les optimisations vectorielles envisagées.

Bibliographie

[1] *The Boost Graph Library : User Guide and Reference Manual.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
Cité page 49.

[2] FullSWOF 2D `http://www.univ-orleans.fr/mapmo/soft/FullSWOF/`, 2011.
Cité page 89.

[3] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).* Addison-Wesley Professional, 2004.
Cité pages 49 et 158.

[4] A. Albrecht. Measuring Application Development Productivity. In I. B. M. Press, editor, *IBM Application Development Symp.*, pages 83–92, Oct. 1979.
Cité page 59.

[5] A. Alexandrescu. *Modern C++ Design : Generic Programming and Design Patterns Applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
Cité page 49.

[6] G. Almasi. PGAS (Partitioned Global Address Space) Languages. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer, 2011.
Cité pages 16 et 47.

[7] Apache Software Foundation. Apache Giraph. `http://giraph.apache.org`.
Cité page 53.

[8] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *J. Parallel Distrib. Comput.*, 68(5) :609–625, May 2008.
Cité page 35.

[9] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level Direct K-way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *J. Parallel Distrib. Comput.*, 68(5) :609–625, May 2008.
Cité page 43.

[10] S. Balay, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.4,

Argonne National Laboratory, 2013.
Cité page 54.

[11] S. Balay, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2014.
Cité page 54.

[12] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.
Cité page 54.

[13] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. Comput.*, 36(5) :570–580, May 1987.
Cité page 41.

[14] J. Bigot. *Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul scientifique.* PhD thesis, INSA de Rennes, Dec. 2010.
Cité pages 18 et 156.

[15] J. Bigot, Z. Hou, C. Pérez, and V. Pichon. A low level component model easing performance portability of HPC applications. *Computing*, Nov. 2013.
Cité pages 18 et 156.

[16] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1) :39–59, Feb. 1984.
Cité page 18.

[17] R. Bisseling. *Parallel Scientific Computation : A Structured Approach Using BSP and MPI.* Oxford scholarship online. OUP Oxford, 2004.
Cité pages 18, 34 et 45.

[18] R. H. Bisseling, B. O. Fagginger Auer, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek. Two-dimensional approaches to sparse matrix partitioning. In U. Naumann and O. Schenk, editors, *Combinatorial Scientific Computing*, Computational Science Series, pages 321–349. CRC Press, Taylor & Francis Group, Boca Raton, FL, 2012.
Cité pages 38 et 40.

[19] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.
Cité page 54.

[20] F. Bouchut. *Nonlinear stability of finite volume methods for hyperbolic conservation laws, and well-balanced schemes for sources*, volume 2/2004. Birkhäuser Basel, 2004.
Cité pages 91 et 154.

[21] S. Breuer, M. Steuwer, and S. Gorlatch. Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. *HiStencils 2014*, page 15, 2014.
Cité page 51.

[22] A. Bukhamsin, M. Sindi, and J. Al-Jallal. Using the Intel MPI Benchmarks (IMB) to Evaluate MPI Implementations on an Infiniband Nehalem Linux Cluster. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 240 :1–240 :4, San Diego, CA, USA, 2010. Society for Computer Simulation International.
Cité page 16.

[23] D. A. Burgess, P. I. Crumpton, and M. B. Giles. A Parallel Framework for Unstructured Grid Solvers, 1994.
Cité page 55.

[24] A. Buss, A. Fidel, H. Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pView. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 261–275, Berlin, Heidelberg, 2011. Springer-Verlag.
Cité page 48.

[25] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL : Standard Template Adaptive Parallel Library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14 :1–14 :10, New York, NY, USA, 2010. ACM.
Cité pages 48 et 154.

[26] U. V. ¸Catalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. 1117 :75–86, 1996.
Cité page 38.

[27] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.
Cité page 43.

[28] U. V. Catalyurek and C. Aykanat. Hypergraph-Partitioning Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Trans. on Parallel and Distributed Computing*, 10 :673–693.
Cité page 38.

[29] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7) :673–693, 1999.
Cité page 144.

[30] Ü. V. Çatalyürek and C. Aykanat. A Fine-Grain Hypergraph Model for 2D Decomposition of Sparse Matrices. In *IPDPS*, page 118, 2001.
Cité page 40.

[31] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3) :291–312, Aug. 2007.
Cité page 47.

[32] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. ZPL : A Machine Independent Programming Language for Parallel Computers. *IEEE Trans. Software Eng.*, 26(3) :197–211, 2000.
Cité page 47.

[33] A. Chan and F. Dehne. CGMgraph/CGMlib : Implementing and Testing CGM Graph Algorithms on PC Clusters. In *International Journal of High Performance Computing Applications*, page 2005. Springer, 2003.
Cité page 49.

[34] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP : Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.
Cité pages 15 et 46.

[35] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10 : An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40(10) :519–538, Oct. 2005.
Cité page 47.

[36] M. Christen, O. Schenk, and H. Burkhart. PATUS : A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, May 2011.
Cité page 54.

[37] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli : A comprehensive overview. ERCIS Working Papers 7, Westfälsche Wilhelms-Universität Münster (WWU) - European Research Center for Information Systems (ERCIS), 2009.
Cité page 51.

[38] J. Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engg.*, 11(4) :29–41, July 2009.
Cité page 52.

[39] M. Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30 :389–406, March 2004.
Cité page 50.

[40] M. I. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation*. PhD thesis, University of Glasgow, 1988. AAID-85022.
Cité page 50.

[41] S. Cordier, H. Coullon, O. Delestre, C. Laguerre, M. H. Le, D. Pierre, and G. Sadaka. FullSWOF Paral : Comparison of two parallelization strategies (MPI and

SKELGIS) on a software designed for hydrology applications. *ESAIM : Proc.*, 43 :59–79, 2013.
Cité page 89.

[42] H. Coullon, J.-M. Fullana, P.-Y. Lagrée, S. Limet, and X. Wang. Blood Flow Arterial Network Simulation with the Implicit Parallelism Library SkelGIS. In *ICCS*, 2014.
Cité page 123.

[43] H. Coullon, M.-H. Le, and S. Limet. Parallelization of Shallow-Water Equations with the Algorithmic Skeleton Library SkelGIS. In *ICCS*, volume 18 of *Procedia Computer Science*, pages 591–600. Elsevier, 2013.
Cité page 89.

[44] H. Coullon and S. Limet. Implementation and Performance Analysis of SkelGIS for Network Mesh-based Simulations. In *Euro-Par*, 2014.
Cité page 112.

[45] P. Crumpton and M. B. Giles. Multigrid aircraft computations using the OPlus parallel library, 1995.
Cité page 55.

[46] Dalissier, E., Guichard, C., Havé, P., Masson, R., and Yang, C. ComPASS : a tool for distributed parallel finite volume discretizations on general unstructured polyhedral meshes. *ESAIM : Proc.*, 43 :147–163, 2013.
Cité pages 42, 43 et 67.

[47] J. Dean and S. Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1) :107–113, Jan. 2008.
Cité page 52.

[48] O. Delestre. *Simulation du ruissellement d'eau de pluie sur des surfaces agricoles/ rain water overland flow on agricultural fields simulation.* PhD thesis, Université d'Orléans, July 2010.
Cité pages 90, 91 et 154.

[49] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *IPDPS*, 2006.
Cité page 35.

[50] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt : A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9 :1–9 :12, New York, NY, USA, 2011. ACM.
Cité pages 55, 100 et 154.

[51] N. Edmonds and A. Lumsdaine. Distributed Compressed Sparse Row, 2010.
Cité page 113.

[52] J. Enmyren and C. W. Kessler. SkePU : a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.
Cité page 51.

[53] B. O. Fagginger Auer and R. H. Bisseling. Abusing a hypergraph partitioner for unweighted graph partitioning. In D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors, *Graph Partitioning and Graph Clustering*, volume 588 of *Contemporary Mathematics*, pages 19–35. AMS, Providence, RI, 2013.
Cité page 147.

[54] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, 21(9) :948–960, Sept. 1972.
Cité page 13.

[55] M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.
Cité page 54.

[56] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
Cité page 16.

[57] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3) :237 – 267, 1976.
Cité page 35.

[58] K. S. George, G. Karypis, and V. Kumar. A New Algorithm for Multi-objective Graph Partitioning. In *In Proceedings of Europar*, pages 322–331. Springer Verlag, 1999.
Cité page 43.

[59] M. Giles, G. Mudalige, and I. Reguly. OP2 Airfoil Example, 2012.
Cité pages 55 et 100.

[60] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly. Performance Analysis of the OP2 Framework on Many-core Architectures. *SIGMETRICS Perform. Eval. Rev.*, 38(4) :9–15, Mar. 2011.
Cité pages 55, 100 et 154.

[61] R. L. Graham. The MPI 2.2 Standard and the Emerging MPI 3 Standard. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 2–2, Berlin, Heidelberg, 2009. Springer-Verlag.
Cité pages 16, 45 et 47.

[62] D. Gregor and A. Lumsdaine. The parallel bgl : A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC*, 2005.
Cité pages 49 et 113.

[63] W. Gropp. MPICH2 : A New Start for MPI Implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag.
Cité page 16.

[64] T. Gschwind and T. U. Wien. PSTL - A C++ Persistent Standard Template Library. In *In Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*, pages 147–158, 2001.
Cité page 48.

[65] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science, New York, USA, 1977.
Cité pages 59, 95, 98 et 129.

[66] B. Hendrickson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. *Parallel Comput.*, 26(12) :1519–1534, Nov. 2000.
Cité page 36.

[67] B. Hendrickson and R. Leland. The Chaco User's Guide : Version 2.0. Technical Report SAND94–2692, Sandia National Lab, 1994.
Cité page 36.

[68] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.
Cité page 54.

[69] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. J. Lang, S. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib : The BSP programming library. *Parallel Computing*, 24(14) :1947–1980, 1998.
Cité pages 18 et 47.

[70] J. Hoberock and N. Bell. Thrust : A Parallel Template Library, 2010. Version 1.7.0.
Cité page 48.

[71] N. Javed and F. Loulergue. OSL : Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In Y. Dou, R. Gruber, and J. Joller, editors, *Advanced Parallel Processing Technologies*, volume 5737 of *Lecture Notes in Computer Science*, pages 436–451. Springer Berlin Heidelberg, 2009.
Cité page 51.

[72] N. Javed and F. Loulergue. Parallel Programming and Performance Predictability with Orléans Skeleton Library. In *International Conference on High Performance*

Computing and Simulation, pages 257–263, Istanbul, Turquie, 2011. IEEE.
Cité page 51.

[73] F. Jedrzejewski. *Introduction aux méthodes numériques*. Springer, Paris, Berlin, Heidelberg, 2000.
Cité pages 19, 32, 89, 91 et 123.

[74] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.
Cité page 54.

[75] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS : A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.
Cité page 52.

[76] Y. Karasawa and H. Iwasaki. A Parallel Skeleton Library for Multi-core Clusters. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 84–91, Sept 2009.
Cité page 51.

[77] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1) :359–392, Dec. 1998.
Cité pages 36, 43, 139 et 147.

[78] G. Karypis, V. Kumar, and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48 :96–129, 1998.
Cité page 35.

[79] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.
Cité pages 13 et 46.

[80] C. F. Kemerer. An Empirical Validation of Software Cost Estimation Models. *Commun. ACM*, 30(5) :416–429, May 1987.
Cité page 59.

[81] M. Kim and K. Candan. SBV-Cut : Vertex-Cut Based Graph Partitioning Using Structural Balance Vertices. *Data and Knowledge Engineering*, 72 :285–303, 2012.
Cité page 138.

[82] D. Kopriva. *Implementing Spectral Methods for Partial Differential Equations : Algorithms for Scientists and Engineers*. Mathematics and Statistics. Springer, 2009.
Cité page 30.

[83] H. W. Kuhn. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming*, pages 29–47, 2010.
Cité page 148.

[84] J. Legaux. *Squelettes algorithmiques pour la programmation et l'exécution efficaces de codes parallèles.* These, Université d'Orléans, Dec. 2013.
Cité pages 50 et 51.

[85] X. Li, S. Ramanathan, and M. Parashar. Hierarchical Partitioning Techniques for Structured Adaptive Mesh Refinement (SAMR) Applications. In *ICPP Workshops*, pages 336–343. IEEE Computer Society, 2002.
Cité page 157.

[86] Z. Li and Y. Song. Automatic Tiling of Iterative Stencil Loops. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGE SYSTEMS*, 26 :2004, 2004.
Cité page 54.

[87] F. Magoulès and F. Roux. *Calcul scientifique parallèle : Cours, exemples avec openMP et MPI , exercices corrigés.* Mathématiques appliquées pour le Master/SMAI. Dunod, 2013.
Cité page 45.

[88] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel : A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.
Cité pages 52 et 154.

[89] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis : an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11 :1–11 :12. ACM, 2011.
Cité page 54.

[90] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, 32(7–8) :590 – 603, 2006. Algorithmic Skeletons.
Cité page 51.

[91] T. J. McCabe. A Complexity Measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
Cité page 59.

[92] W. F. McColl. Scalability, Portability and Predictability : The BSP Approach to Parallel Programming. *Future Gener. Comput. Syst.*, 12(4) :265–272, Dec. 1996.
Cité pages 17 et 45.

[93] J. Merlin and A. Hey. An Introduction to High Performance Fortran. *Sci. Program.*, 4(2) :87–113, Apr. 1995.
Cité page 46.

[94] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly. OP2 : An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012.
Cité pages 55, 100 et 154.

[95] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition : C++ Programming with the Standard Template Library.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
Cité page 47.

[96] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly, 1996.
Cité page 15.

[97] D. Nicol. Rectilinear Partitioning of Irregular Data Parallel Computations. *Journal of Parallel and Distributed Computing*, 23(2) :119 – 134, 1994.
Cité page 41.

[98] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, Aug. 1998.
Cité page 47.

[99] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala : A Comprehensive Step-by-step Guide.* Artima Incorporation, USA, 1st edition, 2008.
Cité page 55.

[100] F. Pellegrini and J. Roman. SCOTCH : A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *HPCN Europe*, pages 493–498, 1996.
Cité pages 36, 43, 139 et 147.

[101] D. M. Pelt and R. H. Bisseling. A medium-grain method for fast 2D bipartitioning of sparse matrices. In *Proceedings IEEE International Parallel and Distributed Processing Symposium 2014*. IEEE Press, 2014.
Cité page 40.

[102] Y. Pinchover and J. Rubinstein. *An Introduction to Partial Differential Equations.* Number vol. 10 in An introduction to partial differential equations. Cambridge University Press, 2005.
Cité pages 25 et 28.

[103] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The TAO of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI 2011, pages 12–25, New York, NY, USA, 2011. ACM.
Cité pages 45, 64 et 158.

[104] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo.

SPIRAL : Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2) :232– 275, 2005.
Cité page 54.

[105] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
Cité page 48.

[106] P. Ridley. Guide to Partitioning Unstructured Meshes for Parallel Computing.
Cité pages 42, 43 et 67.

[107] J. Sanders and E. Kandrot. *CUDA by Example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.
Cité page 18.

[108] K. Schloegel, G. Karypis, and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical report, In Proceedings of Supercomputing, 1998.
Cité page 43.

[109] L. Smith and M. Bull. Development of mixed mode mpi / openmp applications. *Sci. Program.*, 9(2,3) :83–98, Aug. 2001.
Cité page 132.

[110] H. D. Sterck and P. Ullrich. Introduction to Computational PDEs.
Cité pages 25 et 28.

[111] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *IPDPS Workshops*, pages 1176–1182. IEEE, 2011.
Cité page 51.

[112] J. E. Stone, D. Gohara, and G. Shi. OpenCL : A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3) :66–73, May 2010.
Cité pages 18 et 51.

[113] G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pArray. In *Proceedings of the 2007 Workshop on MEmory Performance : DEaling with Applications, Systems and Architecture*, MEDEA '07, pages 73–80, New York, NY, USA, 2007. ACM.
Cité page 48.

[114] G. Tanase, A. Buss, A. Fidel, H. Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 235–246, New York, NY, USA, 2011. ACM.
Cité page 48.

[115] G. Tanase, X. Xu, A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Plist. In *Proceedings of the 22Nd International Conference on Languages and Compilers*

for Parallel Computing, LCPC'09, pages 16–30, Berlin, Heidelberg, 2010. Springer-Verlag.
Cité page 48.

[116] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In L. Fortnow and S. P. Vadhan, editors, *SPAA*, pages 117–128. ACM, 2011.
Cité page 54.

[117] J. Thompson, B. Soni, and N. Weatherill. *Handbook of Grid Generation*. Taylor & Francis, 1998.
Cité page 21.

[118] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8) :103–111, Aug. 1990.
Cité page 17.

[119] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8) :103–111, 1990.
Cité page 45.

[120] B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Rev.*, 47(1) :67–95, Jan. 2005.
Cité pages 38, 39, 137, 145, 146 et 147.

[121] C. Walshaw and M. Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency - Practice and Experience*, 7(1) :17–28, 1995.
Cité page 157.

[122] C. Walshaw and M. Cross. JOSTLE : Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).
Cité pages 42, 43 et 67.

[123] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning For Optimizing Domain Shape, 1999.
Cité page 35.

[124] C. Walshaw, M. Cross, and M. G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Parallel Distrib. Comput.*, 47(2) :102–108, 1997.
Cité page 157.

[125] C. Walshaw, M. Cross, and K. McManus. Multiphase mesh partitioning. *Applied Mathematical Modelling*, 25(2) :123 – 140, 2000. Dynamic load balancing of mesh-based applications on parallel.
Cité pages 35 et 44.

[126] X. Wang, J.-M. Fullana, and P.-Y. Lagrée. Verification and comparison of four numerical schemes for a 1D viscoelastic blood flow model. Technical report.
Cité pages 123, 124, 125 et 155.

[127] T. White. *Hadoop : The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
Cité page 52.

[128] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC : First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.
Cité page 18.

[129] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.
Cité page 45.

[130] A. N. Yzelman and R. H. Bisseling. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation : Practice & Experience*, 24(5) :533–553, 2012.
Cité page 18.

[131] M. Zhou, O. Sahni, K. D. Devine, M. S. Shephard, and K. E. Jansen. Controlling Unstructured Mesh Partitions for Massively Parallel Simulations. *SIAM Journal on Scientific Computing*, 32(6) :3201–3227, 2010.
Cité page 43.

Hélène Coullon

Comparaison et Évolution de Schémas XML

Résumé.
Mots clés.

Comparaison et Évolution de Schémas XML

Abstract.
Keywords.

Laboratoire d'Informatique Fondamentale d'Orléans
Bâtiment 3IA, rue Léonard de Vinci, B.P. 6759
45067 ORLEANS cedex 2, FRANCE