Une librairie de squelettes algorithmiques par blocs sur structure de données 2D

Hélène Coullon
Géo-Hyd et LIFO - Université d'Orléans
Orléans, France
helene.coullon@univ-orleans.fr

Sébastien Limet
LIFO - Université d'Orléans
Orléans, France
sebastien.limet@univ-orleans.fr

Résumé

La plupart des domaines scientifiques actuels disposent d'une quantité de données à traiter de plus en plus importante, posant des problèmes de chargement mais également de calculs sur des machines standard. Cela fait émerger dans beaucoup de domaines scientifiques un fort besoin non seulement en calcul parallèle mais également en distribution de données.

Les librairies de squelettes de programmation parallèle cherchent à proposer une programmation séquentielle à l'utilisateur, tout en lui masquant l'aspect parallèle et complexe qui se cache derrière leur utilisation. Toutefois nous montrerons que l'approche systématique par *éléments*, induit de fortes limitations sur les librairies existantes pour le calcul sur matrices à deux dimensions, et notamment sur de gros volumes de données. Nous présentons dans cet article une nouvelle approche, par blocs, de squelettes de programmation parallèle sur des structures de données distribuées à deux dimensions.

Nowadays, Most sciences have a growing amount of data to compute. It causes load and calculation problems on regular machines. These lacks of power make the need of parallel computing and data distribution grow.

Parallel skeletons librairies aim at giving a sequential way of development to the user, hiding him the parallel and complex aspects of the library itself. However we are going to point out that the systematic approach by *elements* of the current libraries creates a limitation on two dimensions matrices calculations especially on huge data amount. We present in this article a new approach, by blocks, of parallel algorithmic skeletons on two-dimensional data.

1 Introduction

Le parallélisme [18] est un domaine très complexe de l'informatique. Il nécessite à la fois de très bonnes connaissances en programmation générale, des bibliothèques de parallélisme mais également une connaissance solide du fonctionnement des processeurs et de la gestion de la mémoire.

Permettre à des non spécialistes d'accéder aux avantages de la parallélisation est donc un véritable problème. En effet, dans de très nombreux domaines (scientifiques ou non) la rapidité et la quantité de calculs sont devenus des enjeux cruciaux pour lever les verrous que ces domaines rencontrent. Par exemple, des programmes rapides et efficaces offrent des grandes possibilités aux scientifiques pour affiner leurs résultats. Des calculs trop longs, et le manque de temps, impliquent souvent de se contenter de quelques tests, alors qu'un programme plus rapide permettrait de rejouer beaucoup plus de tests dans la même durée, rendant le résultat bien meilleur. Dans d'autres domaines, comme notamment les géo-sciences, le problème se trouve d'avantage sur la quantité de données à traiter en entrée. Cela est notamment dû aux améliorations des moyens d'acquisition (laser aéroporté, images satellites etc.). Là encore le fait de répartir les données et donc la charge sur différents processeurs peut améliorer les temps de traitement et donc la qualité des résultats obtenus.

Il existe plusieurs possibilités pour donner accès au parallélisme à des non-spécialistes, la difficulté étant de trouver une bonne balance entre l'abstraction par rapport au parallélisme et l'efficacité des programmes ainsi écrit. La première possibilité est de réécrire des algorithmes existants, souvent très travaillés et très spécifiques aux métiers qui les utilisent. Ce type de parallélisation d'algorithmes est très long, il implique de comprendre le problème initial, souvent d'un autre domaine scientifique et souvent très complexe, mais aussi d'appréhender les algorithmes séquentiels existants et de comprendre leurs subtilités. Cette parallélisation doit donc s'adresser en priorité à des algorithmes très complexes qu'il est difficile de paralléliser d'une autre façon. La deuxième possibilité est de proposer des solutions de parallélisation automatique à partir de code séquentiels existants, notons par exemple le travail de Ierotheou et al. [10, 11] qui propose une boîte à outils de génération de code automatique OpenMP à partir d'algorithmes séquentiels en Fortran. Toutefois il est montré que cette solution est limitée puisque l'intervention d'un informaticien ou d'un spécialiste du domaine dont on essaie de paralléliser l'algorithme est indispensable pour obtenir une cohérence des résultats et de bonnes performances. La troisième et dernière possibilité se positionne entre ces deux extrêmes. Il s'agit de proposer des environnements ou bibliothèques permettant à l'utilisateur non spécialiste de faire ses propres programmes parallèles sans notion de parallélisme mais en réfléchissant à son problème d'une autre façon, d'une façon facilement parallélisable. Notre travail se positionne dans cette troisième direction.

Notre travail s'est initié par une collaboration avec l'entreprise Géo-Hyd, bureau d'études en hydro-géologie. Les scientifiques de cette société ne possèdent pas de connaissances de spécialistes en programmation parallèle et pourtant ils ont un fort besoin en parallélisme. C'est de cette constatation que sont nés de grandes règles des travaux présentés dans cet article : (1)Apporter un accès simplifié voire transparent au parallélisme, et (2)proposer à un utilisateur de coder de façon séquentielle mais malgré tout obtenir un programme parallèle. Autrement dit, on tend à cacher au maximum les aspects techniques du parallélisme tout en lui permettant d'écrire des programmes efficaces.

Les premiers travaux que nous avons effectués dans cette optique ont abouti à un framework proposant une génération semi-automatique d'application parallèle par composants de type "Ferme de PC" [5, 6] permettant de créer très simplement des programmes parallèles pour des problèmes très locaux d'hydro-géologie, s'apparentant à du calcul paramétrique, en automatisant la division des données à traiter. Ce travail répond à une large classe de problèmes qui se posent dans le domaine de l'hydrologie mais il n'est malheureusement pas suffisant dans tous les cas. En effet, toute une classe de traitement effectués sur des modèles numériques de terrain consistent à effectuer des calculs sur une matrice 2D où chaque point du résultat est le résultat d'un calcul avec les voisins proches de ce point. Ce type de calcul est d'ailleurs très rependu dans de nombreux domaines scientifiques.

Les concepts (1) et (2) énoncés précédemment sont très proches des thématiques des domaines de la programmation par squelettes algorithmiques parallèles, de la programmation stencil parallèle ou des langages dédiés (Domain Specific Language) parallèles. Dans cet article, nous allons présenter la première brique d'une bibliothèque de squelettes de programmation permettant de manipuler de très grandes matrices 2D représentant des modèles numériques de terrain. Cette bibliothèque travaille sur des machines à mémoire distribuée (type grappes de PC) et gère automatiquement et efficacement les communications nécessaires aux calculs.

L'état de l'art de la section 2 présentera en détail ces deux domaines ainsi que notre positionnement. Dans le section 3 nous présenterons la librairie que nous proposons, avec un exemple concret d'utilisation. La section 4 présentera les premiers résultats obtenus et enfin, la section 5 discutera des futurs travaux envisagés.

2 Etat de l'art

Trois grandes familles de solutions pour rendre le parallélisme accessible à des non-spécialistes se rapprochent du travail présenté dans cet article. Il s'agit des squelettes algorithmiques parallèles, la programmation stencil parallèle et les langages dédiés (DSL) parallèles.

Le domaine des Languages Dédiés (DSL) étudie les architectures logicielles permettant de définir des langages dédiés à une problématique spécifique. L'objectif d'un DSL rejoint notre problématique qui consiste à permettre à des spécialistes non informaticiens de développer facilement leurs propres programmes. Dans ce cadre, des travaux ont été effectués pour paralléliser automatiquement des DSL, par exemple autour des langages Scala/Delite [2, 3]. Dans cette architecture, il est possible de définir des DSL en utilisant Scala, cette définition sera ensuite compilée et optimisée par Delite en fonction de l'architecture cible. Notre travail est beaucoup moins généraliste que ce que propose Scala puisqu'on ne permet pas la définition de DSL. En revanche, notre objectif est de traiter le problème des gros volumes de données qui, à notre connaissance, n'est pas spécifiquement étudié dans les DSL. C'est d'ailleurs cet objectif de traitement de très gros volumes de données qui nous oblige, entre autre, à nous intéresser à des systèmes à mémoire distribuée, du fait de l'espace disque et mémoire nécessaire. La plupart des DSL existants s'intéressent à des systèmes à mémoire partagée, ne convenant pas à nos problématiques. Toutefois, il existe des DSL qui utilisent des architectures PGAS (Patitioned Global Address Space). Cette architecture a la particularité d'utiliser de la mémoire distribuée sur un cluster comme si elle était en fait une mémoire partagée, cependant elle impose l'utilisation de barrières de synchronisation tout comme dans OpenMP. Notre solution cherche à éviter ces problèmes de synchronisation. Par contre, à plus long terme, il est envisageable de prévoir le développpement d'un DSL au dessus de notre librairie de squelettes algorithmiques.

En 1989, Muray Cole introduit la notion de squelettes algorithmiques. De cette nouvelle notion naîtra un grand nombre de bibliothèques de squelettes parallèles, permettant un accès au parallélisme simplifié et même parfois transparent par l'utilisation de notions fonctionnelles [9]. On peut noter, par exemple, la notion de *map* qui applique une fonction à un ensemble d'éléments en entrée et écrit le résultat dans un ensemble de sortie, la notion de *reduce* qui applique une fonction de réduction à un ensemble d'éléments en entrée pour ne générer qu'un unique élément en sortie, ou encore *zipwith* qui applique une fonction sur deux ensembles d'éléments en entrée et écrit dans un ensemble d'éléments en sortie. Après avoir réfléchi à son problème par le biais de ces squelettes fonctionnels de façon séquentielle, l'utilisation des librairies permet la création de programmes parallèles de façon transparente.

Des bibliothèques très variées ont vu le jour, s'adressant à toutes sortes d'architectures parallèles mais aussi dans toutes sortes de langages. Parmi les plus connues, nous pouvons citer ASSIST, Calcium, eSkel, Muskel, Muesli, P3L et SKElib, QUAFF, SkeTo et OSL. Il existe quelques particularités dans les librairies de squelettes, notons par exemple la librairie SkePu [7] qui s'intéresse à des architectures hybrides, pouvant à la fois utiliser les ressources graphiques GPU mais aussi les ressources CPU.

Nous avons fait le choix de développer notre solution en C++, tout d'abord pour des raisons de portabilité. Nos travaux doivent pouvoir être utilisés par des scientifiques de domaines très variés, et le C++ est un langage qui se compile sur toute plate-forme et qui est assez souvent connu des scientifiques. De plus, nous n'excluons pas d'intégrer un jour notre travail dans la librairie de squelettes OSL elle même en C++. Enfin, pour nos essais nous travaillons principalement avec la société Géo-Hyd spécialisée en hydro-géologie et qui manipule donc de gros volumes de données géologiques via des SIG (systèmes d'informations géographiques). Or le SIG libre GRASS que nous utilisons beaucoup est écrit en C++, une librairie dans le même

langage facilitera donc son intégration.

Parmi les librairies C++ existantes, on peut principalement noter eSkel [23], Muesli [1,22], QUAFF [8] OSL [12–15] et SkeTo [20,21,25,26]. eSkel et QUAFF sont des bibliothèques très orientées sur l'optimisation de performances. Il est souvent reproché à eSkel notamment de rester trop compliqué d'accès à des non-spécialistes de parallélisme, car nécessitant des notions MPI. QUAFF, de son côté est difficile d'accès car il a été développé sous confidentialité du CEA. OSL est une bibliothèque qui a été mise au point à l'Université d'Orléans, elle offre de nombreux squelettes et bénéficie d'optimisations par expression templates en C++, donnant ainsi de très bonnes performances. Toutefois, OSL est pour le moment uniquement capable de prendre en charge des structure de données 1D comme les vecteurs.

Les deux librairies les plus proches de notre travail sont donc SkeTo et Muesli, toutes deux en C++, et toutes deux donnant accès à des calculs sur structures de données 2D. Sketo dispose d'optimisations C++ la rendant très compétitive au niveau de ses performances, toutefois nos essais ont révélés que des calculs à priori simples sur des grosses matrices à deux dimensions ne sont parfois pas possibles avec SkeTo. En effet, il existe un certains nombre de limites mémoires dans la version actuelle de SkeTo, et l'utilisation successive de la notion de *shift* par exemple opère des copies de matrices. Sur de très gros volumes de données (nos essais ont par exemple lieu sur des matrices de $4000 * 30000$ points), les copies de matrices sont bien évidemment à exclure. Mais là n'est pas le problème le plus important à nos yeux pour les librairies existantes. L'ensemble des librairies de squelettes de programmation parallèle proposent en effet une approche par éléments à l'utilisateur. Cela signifie que l'utilisateur va décrire une fonction qui agit sur un élément (i, j) de la matrice et n'aura pas de visibilité sur le reste de la matrice. Il devient alors compliqué, par exemple, de faire de simples calculs nécessitant les huit voisins autour de l'élément (i, j). Dans ce cas il faut faire appel à un certains nombre d'autres squelettes, notamment des *shift* qui vont avoir la faculté de décaler la matrice. Par la suite, l'imbrication d'appels de différents squelettes permettent ce type de calculs, mais à quel prix ? Un calcul de 8-connexités est un concept simple dans le calcul matriciel, mais par l'utilisation des librairies de squelettes, le problème devient très complexe et peu intuitif.

L'autre type de programmation ayant un fort lien avec nos travaux, est le domaine de la programmation Stencil. Le principe de cette programmation est de proposer des fenêtre de calcul dans une matrice et d'appliquer un calcul sur cette fenêtre. On retrouve dans ce type de programmation l'aspect fonctionnel, le travail de Gabriele Keller et al. [17, 19] est d'ailleurs entièrement mis au point dans le langage fonctionnel Haskell. Toutefois que ce soit dans les travaux de Gabriele Keller et al. ou dans les travaux de Shoaib Kamil et al. [16], la programmation stencil ne semble s'intéresser qu'à des architectures multi-cœur et non à des architectures de type grappe de PC comme dans notre cas, de plus nous pouvons noter que ce type de programmation propose une vision moins généraliste que les squelettes algorithmiques. En effet, la programmation stencil ne s'intéresse qu'à un type de problèmes, à savoir un calcul sur un certain voisinage. Bien que cet article présente, lui aussi, des résultats sur un voisinage, l'objectif de notre bibliothèque est plus large et ne vise pas à s'intéresser à un seul type de calcul. C'est pourquoi nous estimons notre travail plus proche de la notion de squelettes algorithmiques bien que nous n'entrions pas réellement dans les concepts fondamentaux décrits par Muray Cole [4, 24].

Nous proposons une nouvelle approche pour les squelettes, nous cherchons à conserver pour l'utilisateur le fait intuitif qu'une matrice est un ensemble de points qui disposent de deux coordonnées X, Y et qu'il est difficile pour un très grand nombre de calcul matriciels de se limiter à l'accès à un élément. Pour une utilisation simple du squelette, l'utilisateur doit pouvoir, tout comme dans un algorithme séquentiel sur une simple matrice non distribuée, se déplacer dans la

matrice et avoir accès de façon transparentes aux éléments. Cette approche est présentée dans la section suivante. Notre solution propose un nouvel angle de vue pour les squelettes sur matrices à deux dimensions, toutefois elle est moins étoffée pour le moment que les librairies existantes. On peut noter par exemple que la plupart des librairies existantes disposent d'optimisations de performances. Notre librairies n'offre pas encore ce type d'améliorations, toutefois, notre nouvelle vision rend le calcul sur matrice rapide et compétitif, et de plus l'utilisation de la librairies est beaucoup plus intuitive.

3 Une librairie de squelettes sur matrices 2D par bloc

Le concept intuitif des librairies de squelettes algorithmiques sur une matrice est le suivant : le programmeur définit (de façon séquentielle) une fonction $f : E \longmapsto E$ où E est l'ensemble des éléments d'une matrice m. Ensuite le squelette va se charger de l'exécution en parallèle de la fonction f sur tous les éléments de m qui sera distribuée sur la machine parallèle. Dans ce contexte, la fonction f n'a donc accès qu'à un seul élément et n'a pas de visibilité sur le reste de la matrice.

Notre librairie *SkelGIS* propose une approche différente où l'on va demander à l'utilisateur de définir de façon séquentielle une fonction $f : M \longmapsto M$ où M est une matrice 2D. Le squelette de programmation va effectuer une distribution par blocs de la matrice générale de façon transparente pour l'utilisateur et la fonction f sera appliquée en parallèle sur chacun des blocs. Évidemment, comme nous le verrons plus loin, l'application de la fonction f nécessitera souvent des communications qui seront transparentes pour le programmeur. Dans notre librairie, le programmeur dispose pour programmer sa fonction f d'un ensemble d'outils permettant de manipuler la matrice distribuée de façon séquentielle. L'utilisateur code donc une fonction f sur une matrice distribuée tout comme si il codait une fonction séquentielle sur une matrice standard.

Nous allons décrire en détail dans cette section la distribution de la matrice ainsi que les outils de manipulation mis à disposition de l'utilisateur par la librairie SkelGIS.

3.1 Structure de donnée 2D distribuée

Une librairie de squelettes algorithmiques est toujours basée sur une ou plusieurs structures de données sur lesquelles appliquer les squelettes. En parallèle il faut donc une structure de données distribuée sur les différents processeurs, chacun conservant une partie indépendante des données dans sa mémoire.

Nous avons mis en place dans notre librairie une matrice à deux dimensions distribuée *DMatrix*. Lorsque l'utilisateur appelle le constructeur de la matrice distribuée, chaque processeur construit une matrice générale, en fait une décomposition puis ne charge que les données qui le concerne. Le chargement des données dépend donc de son numéro de processeur MPI et chaque processeur chargera une partie différente tout en ayant la même vision générale de la matrice. En d'autres termes, chaque processeur ne charge que la partie des données qu'il doit gérer mais *sait* où se place ce bloc dans la matrice générale. La figure 1 illustre les divisions de matrice effectuées dans le cas de quatre, huit ou seize processeurs MPI.

La division effectuée par notre constructeur de matrice distribuée est régulière. Chaque processeur aura la même taille à traiter. De plus, le découpage est idéal en largeur et longueur. Toutefois, si le nombre de points dans la matrice de départ n'est pas divisible par le nombre de processeurs, certains processeurs auront des données vides dans leur zone à traiter.

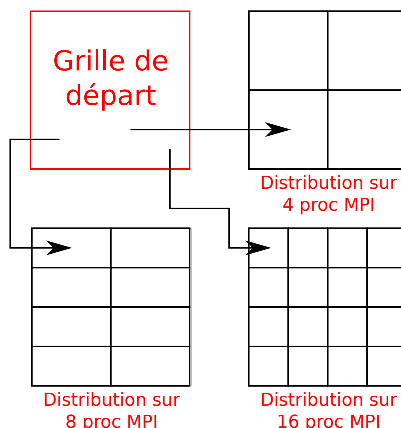

FIGURE 1 – Construction de la matrice distribuée

3.2 Squelettes et outils d'utilisation

Après construction de la matrice distribuée, l'utilisateur doit avoir la possibilité d'appliquer un certain nombre d'actions dessus. La librairie propose pour le moment deux squelettes qui ont la particularité de prendre en compte un recouvrement aux bordures. En effet, le fait de donner la liberté d'accès à l'utilisateur et le fait de lui permettre de pouvoir utiliser un voisinage d'un élément de la matrice, contrairement aux librairies existantes, implique un certain recouvrement de données entre les processeurs. Nous avons appelés nos squelettes *map_recovery* et *zipwith_recovery*.

Tout comme les notions fonctionnelles *map* et *zipwith*, il s'agit d'appliquer dans le premier cas une fonction f à une matrice distribuée d'entrée m, et dans le deuxième cas d'appliquer une fonction f à deux matrices d'entrée. La fonction f en question est écrite par l'utilisateur et est codée de façon séquentielle. Afin que l'exécution en parallèle de la fonction f sur chacun des blocs se passent passe bien, le programmeur devra utiliser les méthodes d'accès à la matrice fournies par notre librairie. La section suivante décrit un exemple simple et complet de l'utilisation de la librairie, et la description d'une fonction y est détaillée. La librairie propose pour le moment quatre outils pour la manipulation de la matrice distribuéeque nous allons décrire.

Lorsque l'on effectue un calcul sur une matrice en programmation séquentielle, le plus important est le parcours des éléments de la matrice. Les librairies de squelettes par élément, cachent justement ce parcours de la matrice et ne donne accès qu'à l'élément courant. Nous avons fait le choix de laisser l'utilisateur faire ce parcours de matrice, tout comme il le ferait en séquentiel et d'appliquer les traitements qu'il souhaite dessus, ceux-ci pouvant dépendre des autres éléments. Le parcours de la matrice distribuée est transparent pour l'utilisateur qui a l'impression de parcourir une matrice séquentielle standard.

Pour parcourir la matrice distribuée, nous avons mis au point un itérateur. Son fonctionnement est le même qu'un itérateur de la librairie standard C++. Le listing 1 montre la déclaration de l'itérateur à l'aide de la méthode *begin* qui renvoie un itérateur sur le premier élément de la matrice m, l'obtention de l'itérateur suivant à l'aide de la méthode *getNextIterator*, et enfin la méthode *end* qui retourne l'itérateur sur le dernier élément de la matrice distribuée.

```
DMatrixBase<float >::iterator debut = matrice->begin();
DMatrixBase<float >::iterator next = matrice->getNextIterator();
DMatrixBase<float >::iterator fin = matrice->end();
```

Listing 1 – Manipulation de l'itérateur sur matrice distribuée

Cet itérateur est la base de deux autres fonctionnalités de la librairie, la première est la possibilité d'avoir un accès direct aux huit valeurs voisines de l'itérateur courant, permettant une simplicité d'utilisation pour des calculs de voisinages simples (Listing 2). C'est dans ce type d'accessions que le recouvrement de nos squelettes prend tout son sens, en effet pour les éléments au bord de la matrice du processeur i il faut avoir les valeurs des processeurs voisins (Voir figure 2).

```
DMatrix<float >::neighbors nghb = matrice->get8Neighbors();
```

Listing 2 – Obtention des huit voisins de l'élément courant

La seconde fonctionnalités en lien direct avec l'utilisation de l'itérateur est une méthode permettant d'attribuer une valeur à l'élément courant pointé par l'itérateur (Listing 3).

```
matrice->setValue(valeur);
```

Listing 3 – écrire une valeur dans l'itérateur courant

Pour finir, notre librairie propose un dernier outil, permettant de façon libre à l'utilisateur d'obtenir la valeur ou de donner une valeur à un élément (i, j) de la matrice distribuée. Si l'utilisateur donne des valeurs (i, j) en dehors de la matrice et de la zone de recouvrement une exception est envoyée. Nous avons couplé cette fonctionnalité avec la possibilité de récupérer les indexes de l'itérateur courant. Le listing 4 montre ces deux dernières fonctionnalités.

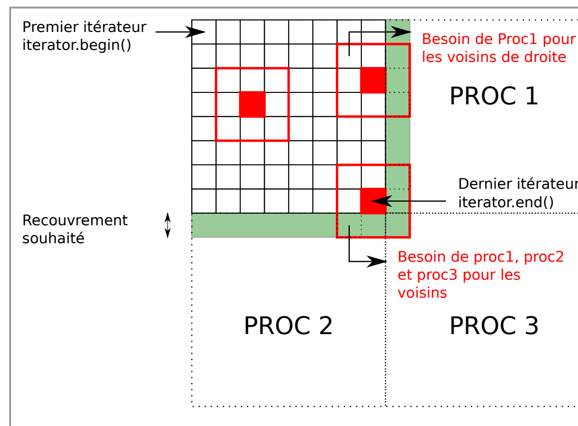

FIGURE 2 – Opérateurs possibles sur la matrice distribuée

```
int i,j;
matrice->getIndexIterator(i,j);
matrice->get(i,j);
matrice->set(i,j,valeur);
```

Listing 4 – écrire ou lire une valeur pour l'élément (i,j)

3.3 Exemple d'utilisation

Nous allons expliquer, dans cette section, la problématique que nous avons traitée pour les résultats présentés par la suite. Nous allons également expliquer comment résoudre ce problème avec la librairie proposée.

Le problème auquel nous nous sommes intéressés est un problème d'hydrologie. A partir d'un fichier représentant une grille de relevés de hauteur de terrain, appelé MNT (Modèle Numérique de Terrain), il s'agit de déterminer en chaque point de la grille vers laquelle des huit directions voisines l'eau s'écoulerait en prenant pour hypothèse qu'elle s'écoulera vers le voisin le plus bas. Ce calcul est appelé directions d'écoulements.

Dans les faits ce problème est simple à appréhender, pour l'élément courant de la matrice il faut obtenir les valeurs des huit voisins et retenir la valeur la plus petite, dans notre cas, si plusieurs voisins correspondent à la hauteur minimale, on en choisira un arbitrairement. Il suffit ensuite d'associer une direction d'écoulement à un entier et de l'inscrire dans la grille de sortie. La figure 3 montre le principe de cet algorithme.

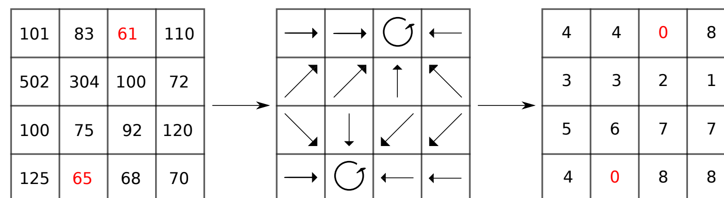

Figure 3 – Calcul des directions d'écoulements

Les MNT sont en fait des matrices à deux dimensions sur lesquels se basent une grande partie des calculs géo-hydrologiques, et ils peuvent représenter de très gros volumes de données. Les MNT sont de plus en plus difficiles à traiter en des temps raisonnables, ce phénomène accentué par les techniques d'acquisition de plus en plus précises (laser aéroportés, données satellites etc.). La parallélisation de leur traitements est donc un point essentiel pour permettre de prendre en charge ces volumes de données.

Pour résoudre ce problèmes avec notre librairie, il y a peu de manipulations à effectuer. Il faut tout d'abord créer la matrice distribuée. Pour cela on fait appel au constructeur d'objet *DMatrix* (Listing 5). Cette étape va créer la matrice distribuée c'est à dire que chaque processeur va construire la division de la matrice et charger dans sa mémoire la partie de la matrice qui le concerne.

```
DMatrix<float> * m = new DMatrix<float>(FichierACharger);
```

Listing 5 – Construction matrice distribuée

L'utilisateur doit ensuite définir le calcul qu'il veut appliquer sur la matrice. Pour cela il a besoin de l'itérateur sur la matrice distribuée et de la fonction d'obtention des huit voisins pour l'élément courant. Pour avoir accès à ces outils l'utilisateur doit définir le code de sa fonction dans une structure prédéfinie puis simplement écrire son code séquentiel. Le listing 6 montre cette étape de programmation. Notons que la valeur -9999 indique l'absence de donnée de hauteur dans le MNT.

```
//-----------------structure predefinie---------------------//
struct ecoulement
{
        void operator()(DMatrixBase<float> * input,DMatrixBase<int> * output) const
        {
//------------------------------------------------------------//
//-------------------Code utilisateur-----------------------//
                DMatrixBase<float>::iterator it = input->begin();
                DMatrixBase<int>::iterator it2 = output->begin();
                while(it!=NULL)
                {
                        int index=0
                        float min = *it;
                        DMatrix<float>::neighbors nghb = input->get8Neighbors();
                        //found the minimum of the 8 neighbors
                        for(int i=0;i<8;i++)
                        {
                                if(nghb[i]!=-9999 && nghb[i]<min)
                                {
                                        index=i+1;
                                        min=nghb[i];
                                }
                        }
                        delete nghb;

                        output->setValue(index);

                        it=input->getNextIterator();
                        it2=output->getNextIterator();
                }
//------------------------------------------------------------//
//-----------------fin de structure predefinie-------------------//
        }
}ecou_f;
//------------------------------------------------------------//
```

Listing 6 – fonction de calcul d'écoulements

Enfin l'utilisateur n'a plus qu'à appeler le *map* sur la matrice qu'il a construite avec la fonction d'écoulement en indiquant le recouvrement dont il a besoin pour le calcul (Listing 7). Il obtient un programme parallèle qui, sans le savoir, distribue son MNT sur les processeurs et fait des échanges de bordures entre les processeurs avant d'appliquer son calcul de directions d'écoulement sur tous les éléments.

```
DMatrix<int> * m2 = map_recovery<ecoulement,float,int>(ecou_f,m,1);
```

Listing 7 – Appel au Map sur la matrice distribuée

4 Résultats

4.1 Comparaison avec la librairie SkeTo

Nous avons effectué des essais pour comparer l'utilisation de SkeTo et de notre librairie, sur le problème de 8-connexités expliqué précédemment. Il est toutefois bon de rappeler que nous sommes dans un cas de traitement sur des matrices à deux dimensions qui nécessite l'accès au voisinage de chaque point (i, j) de la matrice. SkeTo n'a pas vraiment été conçu et optimisé pour ce type de calculs. L'objet de notre comparaison n'est pas ici de minimiser l'intérêt de la librairie SkeTo mais de montrer les limites d'une telle librairie et de montrer l'intérêt notre approche pour ce type de traitements.

Nous avons effectué ces tests sur une machine peu puissante possédant deux coeurs et 2Go de RAM, les deux cœurs de la machine ont été exploités. Nous avons effectué le test sur deux jeux de données différents :

- une matrice de taille modeste $(339 * 225)$ qui représente une sous-partie du bassin Loire-Bretagne
- la matrice complète du bassin Loire-Bretagne $(14786 * 10086)$.

L'ensemble des tests ont été joués quatre fois et une moyenne a été faite pour obtenir le résultat indiqué. Enfin, les temps obtenus représentent uniquement les temps de calcul et sans prendre en compte les temps de chargement des données en mémoire. Le tableau 1 montre ces résultats.

Taille des matrices	**SkeTo**	**SkelGIS**
339 * 225	53	14
14786 * 10086	X	13907

TABLE 1 – Comparaison entre SkeTo et notre librairie en millisecondes

Nous pouvons immédiatement noter que sur la deuxième matrice, SkeTo n'a pas été capable de terminer le calcul. Cette seconde matrice peut paraître grande mais dans le domaine des géo-sciences il s'agit d'une petite portion de terrain. C'est pourquoi le test de passage à l'échelle est très important pour ce type de problèmes.

Il est facile d'expliquer la différence de performance sur ce type de calculs. Avec SkeTo ou toute autre librairie de squelettes par élément, il n'est pas simple d'effectuer le calcul de 8-connexité. Le nombre d'appels imbriqués de squelettes est impressionnant pour réussir à traiter ces problèmes. Par exemple pour ce traitement il faut tout d'abord appeler 8 squelettes *shift* pour obtenir les huit voisins de l'élément courant. Puis appeler une suite d'imbrication de squelettes (le listing 8 montre la liste d'appels imbriqués) où les matrices "temp" sont les résultats des *shift*.

```
dist_matrix<Couple> zip = sketo::matrix_skeletons::zipwith(minimum_f,sketo::matrix
    _skeletons::zipwith(minimum_f,sketo::matrix_skeletons::zipwith(minimum_f,sketo
    ::matrix_skeletons::zipwith(minimum_f,temp0,temp1),sketo::matrix_skeletons::
    zipwith(minimum_f,temp2,temp3)),sketo::matrix_skeletons::zipwith(minimum_f,
    sketo::matrix_skeletons::zipwith(minimum_f,temp4,temp5),sketo::matrix_
    skeletons::zipwith(minimum_f,temp6,temp7))),temp8);
dist_matrix<int> ecoulements = sketo::matrix_skeletons::map(get_dist_f,zip);
```

Listing 8 – Succession d'appels à faire en SkeTo pour le calcul d'écoulement

Ces imbrications vont permettre de comparer deux à deux les valeurs des huit voisins et la

valeur de l'élément (i, j), et de retenir la plus petite.

Autrement dit pour le même algorithme que celui décrit dans la section précédente, une librairie de squelettes classique demande huit appels à *map(shift)* puis huit appels imbriqués de *zipwith* et enfin un dernier *map*, là où notre librairie ne demande qu'un appel à un unique *map*. Cela est dû au fait que notre approche procède aux échanges réseaux nécessaires avant le calcul et permet un accès direct aux autres éléments de la matrice à l'intérieur du *map*, ce qui est impossible dans les librairies existantes.

L'autre différence majeure concernant ces résultats, vient de SkeTo lui même qui dans un *shift* fait une copie de l'ensemble de la matrice. La matrice du bassin Loire-Bretagne fait 570Mo, si on multiplie par huit (les huit copies des *shift*), on arrive à plus de 4Go de mémoire vive nécessaire, ce qui est trop important pour la machine de test. Mais comme précisé précédemment, il ne s'agit là que d'un petit exemple. Un bassin comme celui du Niger pèse 4,7Go, avec huit copies de matrices on arriverait à 37,6Go et pour l'Europe à 50,4Go de mémoire vive nécessaire pour effectuer le calcul en SkeTo. Il est donc quasi impossible d'effectuer un calcul de ce type sur de gros volumes données en SkeTo.

4.2 Passage à l'échelle et speedup

Nous allons maintenant décrire les résultats obtenus au passage à l'échelle de notre librairie. Les résultats ont étés obtenus sur la grappe de PC de la société Géo-Hyd contenant huit machines bi-processeurs Intel Xeon Westmere à quatre cœurs avec 24 Go de mémoire vive DDR3. Nous avons donc disposé de 64 cœurs de calcul pour ces tests. Là encore nous avons effectué chaque test quatre fois et avons fait une moyenne des résultats obtenus. Nous avons uniquement pris en compte le temps de calcul pour ces résultats et non le chargement des données en mémoire.

Nous avons fait nos essais sur deux jeux de données. Le bassin Loire-Bretagne utilisé également dans la section précédente et représentant une matrice de $14786 * 10086$ points, ainsi que le bassin du Niger représentant une matrice de $42000 * 30000$ points.

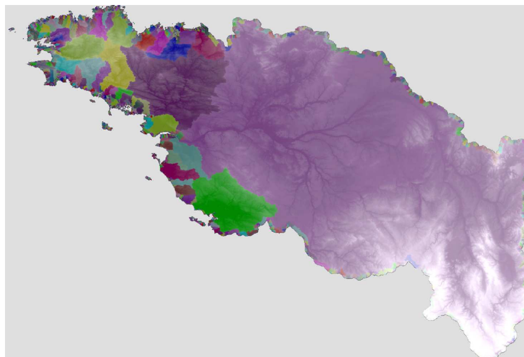

Figure 4 – Bassin Loire-Bretagne

Les figures 5 et 6 présentent les résultats obtenus.

Nous pouvons noter que nos speedup sont linéaires et proches du speedup idéal $x = y$. Toutefois on remarque à l'oeil nu un défaut sur le speedup du bassin Loire-Bretagne, une déviation au passage à quatre processeurs. Cette perte de performance au passage à 4 processeurs sur ces données, est due au fait que le bassin Loire-Bretagne est irrégulier en quantité de données à

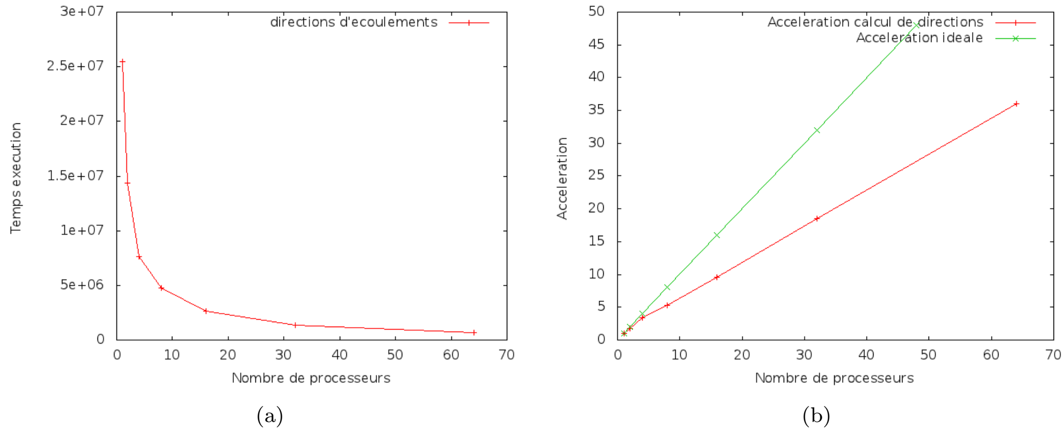

FIGURE 5 – Résultats de directions d'écoulements sur le bassin Loire-Bretagne (a) Courbe des temps d'exécution (b) Courbe d'accélération

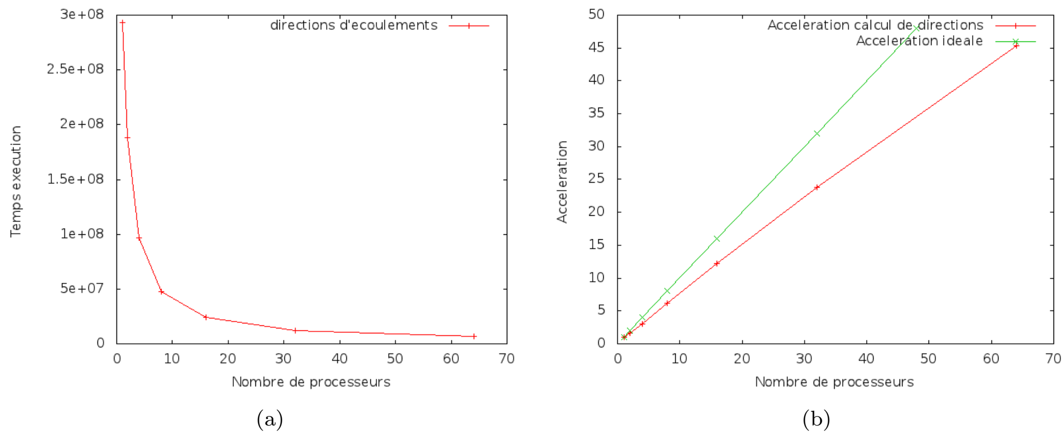

FIGURE 6 – Résultats de directions d'écoulements sur le bassin du Niger (a) Courbe des temps d'exécution (b) Courbe d'accélération

traiter lorsqu'on le divise en quatre. En effet, on voit nettement sur la figure 4 que le sud-ouest du bassin ne possède aucune données à traiter, et par conséquent, une division régulière en grille comme celle effectuée par notre librairie ne va pas donner la même charge de travail aux différents processeurs. Nous n'utilisons donc pas au mieux les capacités de la grappe dans le cas de données très irrégulières en quantité de calculs. En observant en détail les temps d'exécution de chaque processeur, il apparaît de façon plus claire que les différents processeurs ne calculent pas la même quantité de données et n'ont pas la même charge de travail. Ce problème se produit également sur le bassin du Niger, toutefois la différence est moins flagrante sur ces données qui sont plus régulières. Le speedup observé est d'ailleurs meilleur.

5 Conclusion

Nous avons présenté dans cet article une nouvelle approche pour une librairie de squelettes algorithmiques parallèles. En effet, les essais effectués sur certains calculs scientifiques sur des matrices à deux dimensions avec les libraires de squelettes existantes ont fait transparaître que la vision existante n'était pas efficace si le but était de rendre simple l'accès à la parallélisation aux utilisateurs. Une vision fonctionnelle pure avec un accès aux données par élément unique a beaucoup de qualités du point de vue de la conception, de la sémantique claire du concept mais également du point de vue de la vérification de programmes. En revanche cette vision peu rendre certains problèmes, simples à appréhender, très compliqués à mettre en place par ce biais.

Notre librairie de squelettes en est à sa première version et demande à être enrichie. Elle permet cependant, d'obtenir des performances très intéressantes sur des problèmes de calculs avec une notion de voisinage, tout en ne nécessitant qu'un nombre limité d'appels de fonctions. L'utilisateur obtient une vision séquentielle des matrices distribuées par la librairie, et dispose de plus d'outils permettant leur manipulation. L'appel aux squelettes s'apparente vraiment à ce que le programmeur aurait écrit en séquentiel avec un langage de programmation classique ce qui rend l'utilisation de la librairie accessible à des non spécialistes du parallélisme.

Notre librairie ne dispose pour le moment que de deux squelettes algorithmiques *map* et *zipwith*, nous allons la faire évoluer vers d'autres squelettes tels que *reduce* par exemple.

Malgré des performances intéressantes, la librairie ne dispose pas d'optimisations pouvant la rendre encore plus rapide, nous allons nous intéresser à l'optimisation de l'utilisation du cache mais aussi à l'utilisation de registres SSE ou encore aux *expression templates* rendant possible une résolution d'une partie des appels à la compilation.

De plus, une version *hybride* de notre librairie est en cours de développement. Elle proposera un code hybride en MPI et OpenMP. Nous souhaitons, en effet, pouvoir profiter du fait que les machines d'une grappe de PC disposent chacune d'un certain nombre de cœurs disposant d'une mémoire partagée. Dans ce cas certains échanges réseaux peuvent être évités et probablement des gains de performances observés.

Nous ne nous sommes intéressés dans cet article qu'à des problèmes nécessitant un voisinage fixe. Ce problème est proche de la programmation stencil. Nous souhaiterions, d'une proposer à l'utilisateur un moyen de définir le voisinage dont il a besoin pour son calcul. Cette définition permettrait de définir automatiquement la bordure nécessaire au calcul en parallèle. Par ailleurs, pour permettre de traiter des calculs qui ont besoin d'un voisinage non connu à l'avance, nous souhaitons proposer une approche d'obtention de valeurs voisines dynamique. Dans ce cas l'échange n'aurait pas lieu avant l'application des fonctions aux éléments mais en cours de calcul.

Enfin, nous avons identifié une autre classe de problèmes pouvant être résolue par l'utilisation de squelettes algorithmiques. La structure de données serait en revanche très différentes puisqu'il s'agirait d'un arbre de hiérarchie. Dans ce cadre, un calcul impacterait un ensemble d'éléments fils ou père du noeud de l'arbre courant. Ce type de résolution s'appliquerait à des problèmes de transferts, on peut citer par exemple les transports de polluants dans les rivières, les calculs de débits en cas de crues etc.

Références

[1] George Horatiu Botorog and Herbert Kuchen. Efficient parallel programming with algorithmic skeletons. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par,*

Vol. I, volume 1123 of *Lecture Notes in Computer Science*, pages 718–731. Springer, 1996.

[2] K.J. Brown, A.K. Sujeeth, H.J. Lee, T. Rompf, H. Chafi, and K. OLUKOTUN. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[3] Hassan Chafi, Zach Devito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing, 2010.

[4] Murray Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30 :389–406, March 2004.

[5] Hélène Coullon, Audrey Latapie, Sébastien Limet, Emmanuel Melin, Daniel Pierre, Sophie Robert, and Xavier Thomas. Calculs parallèles pour le traitement des gros volumes de données liées aux risques environnementaux. *Ingénierie des Systèmes d'Information (ISI)*, 16(3) :31–54, 2011.

[6] Hélène Coullon, Sébastien Limet, and Emmanuel Melin. A simple framework to generate parallel application for geospatial processing. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & ; Application*, COM.Geo '10, pages 32 :1–32 :4, New York, NY, USA, 2010. ACM.

[7] Johan Enmyren and Christoph 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.

[8] J Falcou, J Serot, T Chateau, and J Lapreste. Quaff : efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7-8) :604–615, 2006.

[9] Kevin Hammond. Parallel functional programming : An introduction. In *International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 1994. World Scientific.

[10] C. S. Ierotheou, H. Jin, G. Matthews, S. P. Johnson, and R. Hood. Generating openmp code using an interactive parallelization environment. *Parallel Comput.*, 31 :999–1012, October 2005.

[11] C. S. Ierotheou, S. P. Johnson, P. F. Leggett, and M. Cross. Using an interactive parallelisation toolkit to parallelise an ocean modelling code. *Future Gener. Comput. Syst.*, 19 :789–801, July 2003.

[12] Noman Javed. *Squelettes algorithmiques méta-programmés : implantations, performances et sémantique*. These, Université d'Orléans, October 2011.

[13] Noman Javed and Frédéric Loulergue. A Formal Programming Model of Orléans Skeleton Library. In Springer, editor, *Parallel Computing Technologies*, LNCS, page to appear, Russie, Fédération De, 2011. Springer.

[14] Noman Javed and Frédéric 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.

[15] Noman Javed, Frédéric Loulergue, Julien Tesson, and Wadoud Bousdira. Prototyping a Library of Algorithmic Skeletons with Bulk Synchronous Parallel ML. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, page to appear, États-Unis, 2011. CSREA Press.

[16] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *In International Parallel and Distributed Processing Symposium (IPDPS*, 2010.

[17] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM.

[18] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[19] Ben Lippmeier and Gabriele Keller. Efficient parallel stencil convolution in haskell. *SIGPLAN Not.*, 46(12) :59–70, September 2011.

[20] Kiminori Matsuzaki and Kento Emoto. Implementing fusion-equipped parallel skeletons by expression templates. In *Proceedings of the 21st international conference on Implementation and application of functional languages*, IFL'09, pages 72–89, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proceedings of the 1st international conference on Scalable information systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[22] Burkhard Monien and Rainer Feldmann, editors. *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Germany, August 27-30, 2002, Proceedings*, volume 2400 of *Lecture Notes in Computer Science*. Springer, 2002.

[23] Anne Benoit Murray, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *In : 11th Intl Euro-Par : Parallel and Distributed Computing, vol. 3648 of LNCS, 761-770, Lisbona*, pages 761–770. Springer-Verlag, 2005.

[24] In Skeletal Parallel, Anne Benoit, and Murray Cole. Two fundamental concepts. In *The International Conference on Computational Science (ICCS 2005) , Part II, LNCS 3515*, pages 764–771. Springer Verlag, 2005.

[25] Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in sketo skeleton library. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 666–677, Berlin, Heidelberg, 2009. Springer-Verlag.

[26] Karasawa Yuki and Iwasaki Hideya. Parallel skeletons for sparse matrices in sketo skeleton library. 49(SIG3(PRO36)) :1–15, mar 2008.