Le génie pour l'industrie

Laboratoire 3

Simulation du transfert de chaleur en 2D d'une plaque chauffante

Département du génie logiciel et des technologies de l'information

Étudiant	Delisle, Pierre-Luc - DELP26029208 Kristof Boucher Charbonneau - BOUK08059309
Cours	LOG645 - Calcul parallèle
Session	Hiver 2017
Groupe	1
Numéro du laboratoire	3
Chargé de cours	Lévis Thériault
Chargé de laboratoire	Kevin Lachance-Coulombe
Date	23 mars 2017

Rapport	/ 50
• Introduction	/ 5
• Analyse	/ 8
• Conception de l'algorithme	/ 12
• Discussion	/ 20
• Conclusion	/ 5
Code source	/ 50
Total	/100

Table des matières

Liste des figures

Liste des tableaux

Liste des équations

Introduction

Dans ce troisième laboratoire du cours de calcul parallèle, l'équipe devait utiliser pour une dernière fois la librairie « MPI ». Contrairement à « OpenMP » employé lors du dernier laboratoire qui a recours aux « *threads* » pour accomplir son parallélisme, « *MPI* » utilise les processus et les communications de type passage de message entre ces derniers. Contrairement aux « *threads* » qui partagent la même mémoire que le processus, un processus possède sa propre mémoire privée. Le passage de message permet donc d'obtenir une valeur d'un autre processus.

Ce laboratoire permet aux membres de l'équipe de mettre en pratique toutes les connaissances apprises sur le parallélisme depuis le début de la session, soit les concepts de communication, d'agglomération et de répartition dans le but de réaliser une simulation de transfert de chaleur en deux dimensions. Le principal objectif de ce laboratoire est de développer une solution qui résout le problème de transfert de chaleur de manière séquentielle et ensuite une solution exploitant le parallélisme dans le but d'augmenter les performances et réduire le temps de résolution du problème.

À la suite de l'implémentation, le présent rapport de laboratoire présentera l'analyse du problème, la conception de l'algorithme séquentiel et parallèle, le résultat des manipulations et la réponse aux questions de ce laboratoire.

Analyse

Dans cette section, une analyse du problème de simulation de chaleur en deux dimensions a été réalisée. Les difficultés liées à la simulation et la formule pour la réaliser ont été étudiées en détail.

Problème

Pour réaliser la simulation de transfert de chaleur en deux dimensions avec la librairie « MPI », il faut considérer quelques aspects. Tout d'abord, la matrice contient des lignes et des colonnes. L'intersection d'une ligne et d'une colonne est appelée : « subdivision ». Chacune de ces subdivisions contient une valeur à un instant « k », où « k » est le numéro de l'itération. Pour calculer la valeur de la subdivision, il faut préalablement avoir la valeur au-dessus, en dessous à gauche et à droite. La figure suivante illustre une matrice contenant neuf colonnes et neuf lignes.

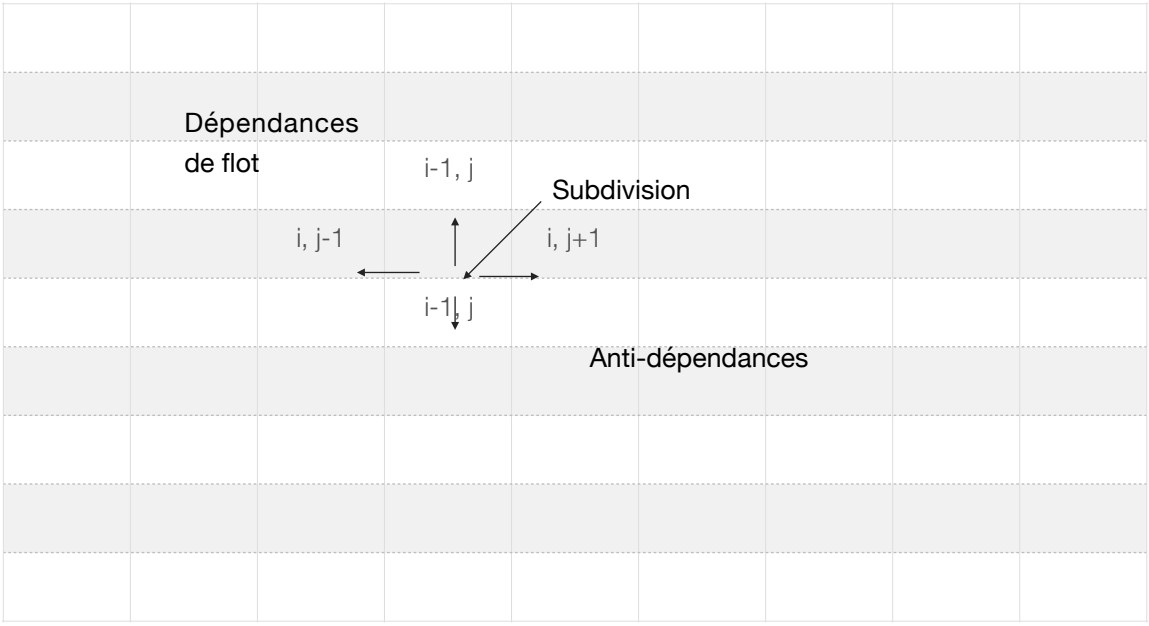


Figure 1 : matrice d'exemple comprenant neuf lignes et neuf colonnes (LOG645, 2017, S.P.)

Dans l'illustration précédente, on voit bien que pour obtenir la valeur de la subdivision, la valeur de chacun des voisins est requise. De façon plus formelle, la dépendance au voisin du dessus et du voisin de gauche est appelée «dépendance de flot» alors que la dépendance à droite et en dessous est appelée « anti-dépendance ». En parallélisme, les dépendances de flots sont les dépendances qui doivent être brisées, car le processus qui traite la subdivision doit attendre de recevoir la valeur de ces dépendances pour continuer son traitement.

Formule

La simulation de transfert de chaleur suit la formule d'Euler suivante :

$$\frac{\delta T(x,y,t)}{\delta t} = C \cdot \left(\frac{\delta^2 T(x,y,t)}{\delta x^2} + \frac{\delta^2 T(x,y,t)}{\delta y^2} \right)$$

Équation 1 : formule d'Euler (LOG645, 2017, S.P.)

Afin de faciliter le problème, cette formule a été simplifiée pour éliminer les dérivées. De plus, certaines hypothèses ont été faites pour simplifier encore plus le problème. Par exemple, la conductivité thermique est égale à 1 sur toute la plaque et les valeurs aux bordures de la matrice sont toutes égales à 0. La formule discrète est la suivante :

$$U(i,j,k+1) = \left(1 - 4 \cdot \frac{t_d}{h^2}\right) \cdot U(i,j,k) + \left(\frac{t_d}{h^2}\right) \cdot \left[U(i-1,j,k) + U(i+1,j,k) + U(i,j-1,k) + U(i,j+1,k)\right]$$

Équation 2 : formule d'Euler simplifiée (LOG645, 2017, S.P.)

Dans la section précédente, la notion de dépendance a été abordée. Les parties en rouge de la formule précédente représentent les dépendances. La première et la troisième partie en rouge représentent les dépendances de flots : il faut avoir la valeur du dessus («i-1») et la valeur de gauche (« j-1»). Les deux autres parties de l'accolade carrée représentent les anti-dépendances.

Finalement, la matrice est initialisée selon la formule suivante :

$$U(i,j,0) = i(n-i-1) \cdot j(m-j-1)$$

Équation 3 : formule d'initialisation de la matrice (LOG645, 2017, S.P.)

Où la valeur de chaque subdivision est dépendante de nombre de ligne et de colonne total ainsi que de l'index de la ligne et de l'index de la colonne actuel.

Conception de l'algorithme

Séquentielle

L'algorithme séquentiel a donné du fil à retordre à l'équipe au début du laboratoire. C'est à la suite de la lecture de *Patterns for Parallel Programming* de Mattson que nous avons trouver une solution optimale. L'algorithme séquentiel est très simple. Il ne s'agit que de trois boucles imbriquées. La première itère sur le nombre d'itérations (*timeStep*); c'est le nombre de fois que la matrice est calculée dans son entièreté. La deuxième boucle *i* parcourt toutes les lignes, alors que la troisième boucle *j* parcours toutes les colonnes. Ainsi, le calcul débute à la coordonnée (0,0) et se dirige vers la droite ligne après ligne. Ainsi, les valeurs nécessaires au calcul pour une coordonnée (*i, j*) (dépendances de flot) sont calculées préalablement. Un tour de passe-passe est nécessaire toutefois pour bien optimiser la chose. En effet, trois matrices doivent être créées préalablement. Une matrice *matrixUk* contient les éléments à l'itération *k*, alors que la matrice *matrixUkp1* contient la nouvelle valeur calculée. À la fin de chacune des itérations sur *k*, un échange de pointeur est fait pour que la matrice *matrixUk* contienne toutes les nouvelles valeurs calculées. Comme nous le verrons, cette même opération sera également utilisée dans la partie parallèle.

Sous forme de pseudo-code, l'algorithme donne ceci :

Tableau 1 : pseudo-code de l'algorithme séquentiel de transfert de chaleur (LOG645, 2017, S.P.)

Pseudo-code
Let matrixUk, matrixUkp1, matrixTemp be 2-dimensional arrays.
Allocate memory for each matrices and fill with zeros.
For all i
For all j
matrixUk = matrixUkp1 = (i * (lineNumber - i - 1)) * (j * (columnNumber - j - 1));
while k <= number of steps
For all i
For all j
matrixUkp1[i] [j] = Euler's heat transfer formula (discretized)
Copy matrixUkp1 in matrixUk.

Le code explicite peut être retrouvé dans la première annexe *Annexe 1 — Principale partie du code séquentiel.*

Parallèle

L'algorithme parallèle fut un réel défi pour l'équipe puisque pour réaliser la simulation du transfert de chaleur, il fallait considérer les quatre dépendances. Les anti-dépendances n'ont pas causé de problèmes lors de la première itération, car la valeur de celle-ci n'était pas nécessaire. Par contre, lors des itérations subséquentes, ces deux anti-dépendances se sont transformées en dépendances de flot et ont rajouté une communication supplémentaire.

Partitionnement, agglomération, répartition

Afin de briser les dépendances de flot, l'équipe a décidé de suivre le patron de conception *Geometric Decomposition Pattern* [1] (Mattson, 2004). Ce patron peut être utilisé de deux manières différentes; on peut effectuer une décomposition 1D ou 2D. La décomposition 1D étant déjà suffisamment complexe à réaliser, c'est cette option qui fut choisie pour ce laboratoire. Dans ce type de décomposition, on découpe la matrice en zones plus petites. Ces zones peuvent comporter une ou plusieurs lignes. Le schéma ci-dessous exprime visuellement ce découpage sur une matrice 10x10 qui serait exécuté sur quatre processeurs.

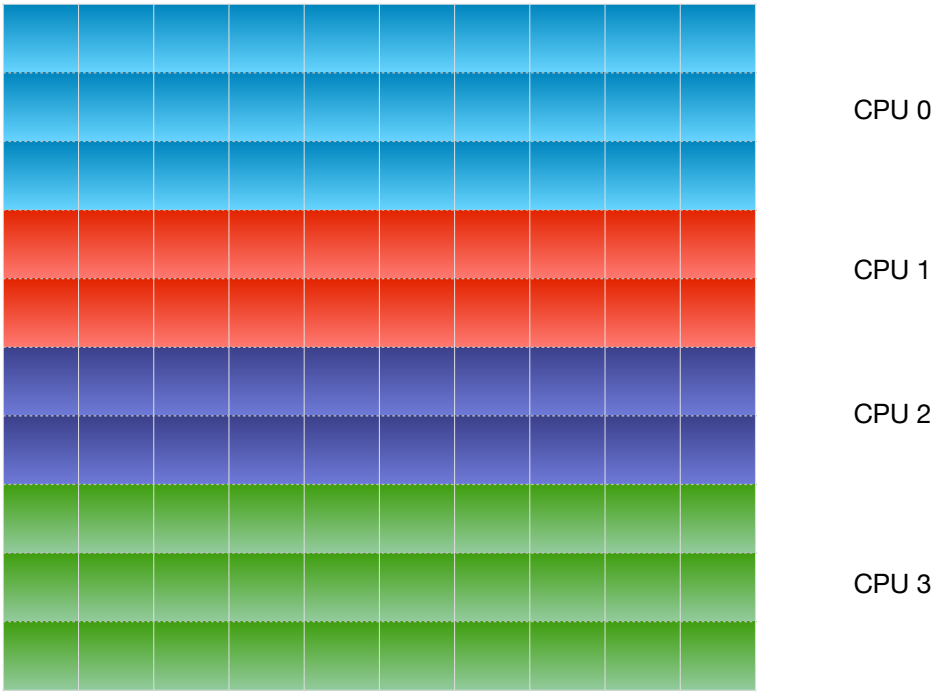


Figure 2 : Décomposition géométrique 1D (LOG645, 2017, S.P.)

L'algorithme de décomposition s'occupe de découper la matrice et d'attribuer à chaque processus la ligne de départ et la ligne de fin que celui-ci se doit de calculer. Pour que cela fonctionne, il

[1] *Patterns for Parallel Computing*, Mattson, p.79-97

faut évidemment que la matrice ait un nombre égal ou supérieur de lignes au de nombre de processeurs sur lesquels on fait l'attribution de ces lignes.

La répartition de la charge reste tout de même très semblable pour tous les CPUs. Le but de la décomposition 1D est d'avoir un équilibrage quasi parfait de la charge entre les processeurs.

Communication

Le patron de décomposition géométrique inclut la notion de *ghosting*. Suivant l'idéologie Single Program Multiple Data (SPMD), on doit s'assurer que chaque processeur a préalablement *toutes* les données nécessaires pour effectuer le travail demandé, en l'occurrence, le calcul d'échange de chaleur selon Euler. Toutefois, nous avons évoqué la présence de quatre dépendances dans la discrétisation d'Euler. Pour chaque ligne à traiter, il nous faut la ligne au-dessus et la ligne en dessous de celle actuellement traitée. Si l'on se fie au schéma précédent, il est impossible pour le CPU 2 d'avoir les données de la ligne supérieure, cette ligne appartenant à la mémoire privée du CPU 1. Il nous faut alors dupliquer l'information de ces lignes de façon à ce que le CPU 2 ait une copie dite *ghost* de la ligne au-dessus. Le schéma ci-dessous illustre cette notion de *data ghosting* :

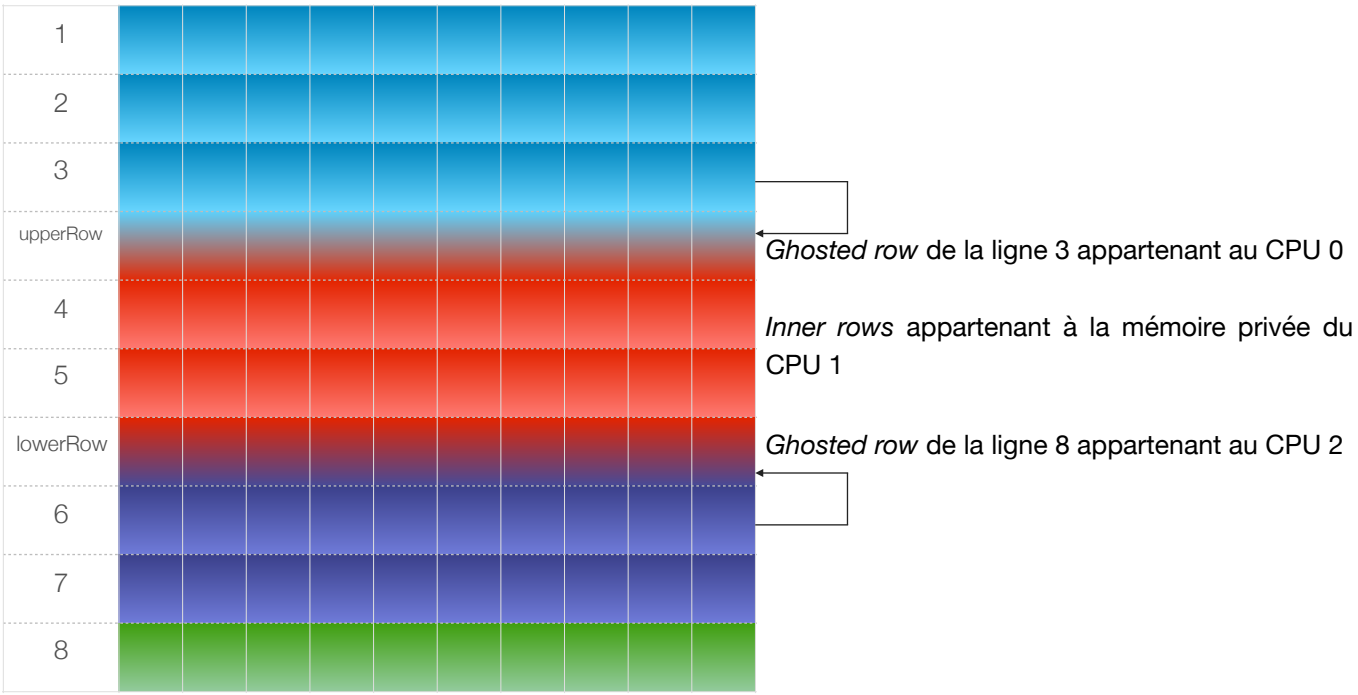


Figure 3 : *Data ghosting* avec focalisation sur le CPU 1 (LOG645, 2017, S.P.)

En ayant réalisé une telle agglomération, il est possible de minimiser le nombre de communications entre les processus. Le processeur 1 a désormais non seulement une copie des données de la dernière ligne du processeur 1 (*matrixRow_struct.upperRow* dans le code source), mais également une copie de la ligne 8, ligne appartenant au CPU 2 (*matrixRow_struct.lowerRow*). De cette

façon, le CPU 1 peut effectuer les calculs sur ses deux lignes pour une itération *k* donnée puisque toutes les dépendances sont désormais accessibles directement dans sa mémoire privée.

Tout comme l'algorithme séquentiel, on doit persister les données à travers une itération. Suivant le même principe, trois structures ont été créées, soit *u_struct, ukp1_struct* et *temp_struct.* De cette manière, on peut copier à la fin de l'itération *k* le contenu de *ukp1_struct* dans *u_struct.*

Il faut par contre à chaque itération échanger les données présentes dans chaque *ghost row* pour que chaque processeur ait l'information de l'itération précédemment complétée. Ainsi, chaque CPU doit échanger sa première ligne et sa dernière ligne avec son processeur voisin, comme le schéma ci-dessous l'exprime :

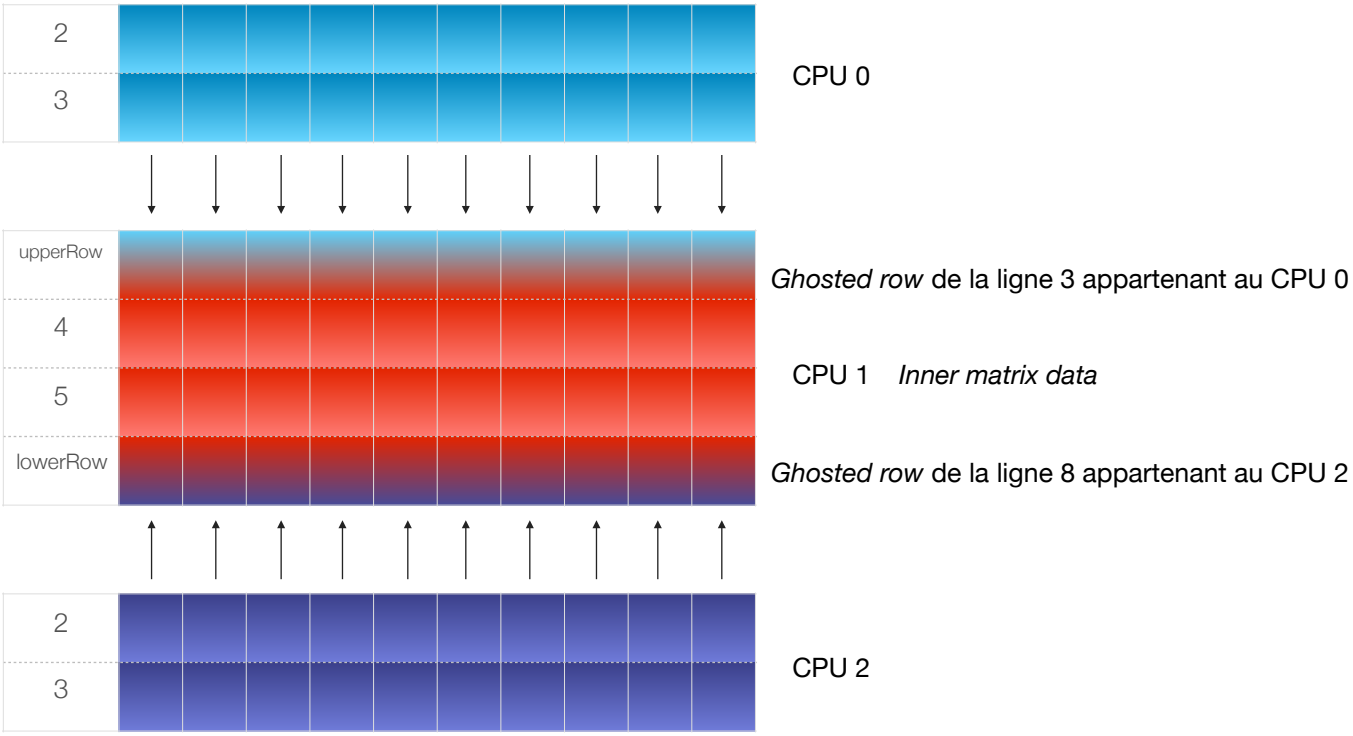


Figure 4 : Communications interprocessus avec *data ghosting* (LOG645, 2017, S.P.)

Il en va de même également pour le processeur 1 qui doit envoyé la ligne 4 dans la ligne *ghost* du processeur 1 et la ligne 5 au processeur 2. Il faut ainsi, à chaque itération, deux envois et deux réceptions pour chaque processeur se trouvant à l'intérieur de la matrice. Pour les processeurs aux extrémités, ils n'ont qu'une seule ligne à envoyer. La surcharge logicielle est toutefois limitée puisque dans le même message nous passons toute l'information de la ligne. Ainsi, une communication est relativement peu coûteuse puisqu'elle peut transmettre toute la largeur de la matrice versus une communication qui ne transmet qu'une ou deux valeurs.

Algorithme et flot d'exécution

L'algorithme, contrairement à l'implémentation en *C*, est relativement peu complexe. Il peut se résumer dans les lignes de pseudo-code de haut niveau ci-dessous :

Tableau 2 : pseudo-code de l'algorithme parallèle de transfert de chaleur (LOG645, 2017, S.P.)

Pseudo-code
Let u_struct and ukp1_struct be structures that maps ghosted rows and inner matrix data.
Make a 1D domain decomposition to find starting line and ending line of each CPUs.
Find upper and lower neighbor.
Initialize u_struct structure with values of inner matrix data.
While k <= number of steps
Send upper row of the inner data to top neighbor.
Send lower row of the inner data to bottom neighbor.
Receive data from top neighbor and place data into upper ghosted row.
Receive data from lower neighbor and place data into lower ghosted row.
Calculate heat transfert.
Copy ukp1_struct in u_struct.

Ce pseudo-code de haut niveau dicte l'allure de l'exécution de notre algorithme parallèle. En premier lieu, deux structures sont créées. La décomposition 1D est ensuite effectuée. L'algorithme de la décomposition fut trouvé en Fortran dans le livre de *Using MPI - Third Edition.* Une traduction de Fortran vers *C* fut par la suite trouvée, la source étant citée dans le code source. Le code source de cette décomposition, pièce maitresse de notre algorithme, peut être trouvé à l'annexe *Annexe 2 — Décomposition géométrique 1 D*.

Il suffit ensuite de trouver les voisins du haut et du bas du processeur actif. On initialise la structure avec les lignes préalablement trouvées par l'algorithme de décomposition. Par exemple, pour le processeur 1 dans la figure 3 ci-haut, l'algorithme de décomposition retournerait *start_y = 4* et *end_y = 5*. Ainsi, la structure *u_struct* contiendrait deux lignes internes, soit les lignes 4 et 5 de la matrice. L'algorithme d'échange des données s'occupe quant à lui de remplir les *upperRow* et *lowerRow* de la structure. Une fois cette dernière entièrement initialisée et pourvue de toutes les données nécessaires au calcul, on peut lancer celui-ci. On copie ensuite la nouvelle structure *ukp1_struct* contenant les nouvelles données dans *u_struct* afin de repartir le calcul à l'itération suivante.

Évidemment, la portion calcul s'adapte à cette structure de donnée qui contient essentiellement des tableaux à une dimension. Somme toute, l'allure du calcul demeure la même, seulement l'accès aux données est différent par rapport à la solution séquentielle.

Discussion

Résultats avec h=0.1, td=0.0002, np=200

Pour une matrice 15x10 pour 200 itérations, le résultat du temps de calcul sur le serveur *log645-srv-1* est le suivant :

Tableau 3 : temps d'exécution pour une matrice de 15x10 (LOG645, 2017, S.P.)

Type d'exécution	Temps d'exécution
Séquentiel	0.255574
Parallèle	0.056672

Comme on peut le constater, le temps d'exécution en parallèle est beaucoup plus bas. Voici les données sur l'accélération et l'efficacité de cette exécution.

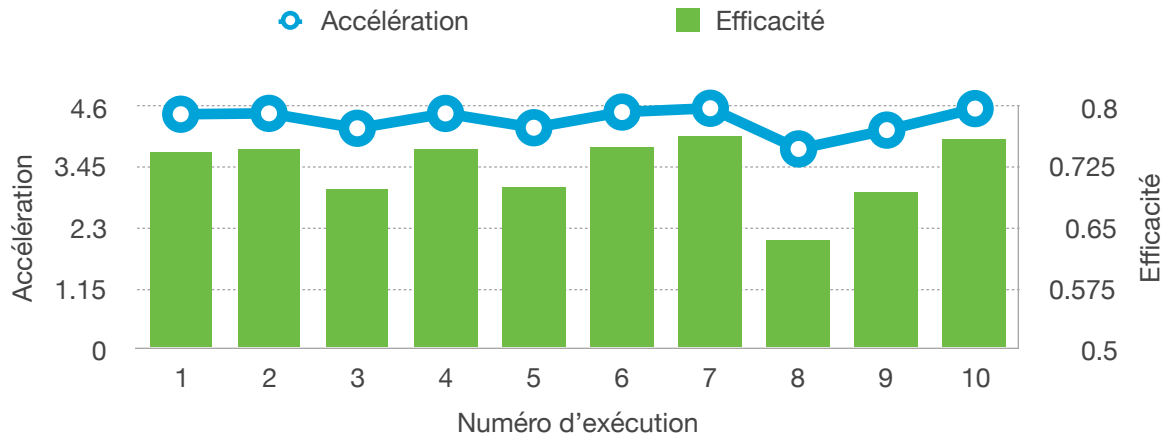
Tableau 4 : Accélération et efficacité pour une matrice de 15x10 (LOG645, 2017, S.P.)

Statistique	Temps d'exécution
Accélération	4.509716
Efficacité	0.751619

Vu la très petite taille du domaine, seulement 6 processeurs au maximum peuvent être utilisés pour résoudre le problème en parallèle. L'algorithme développé basé sur la décomposition géométrique permet d'exploiter tout son potentiel dans des domaines beaucoup plus grands. Il s'agit néanmoins d'un bon départ pour un domaine aussi petit.

Répétabilité du résultat

Voici les données recueillies à la suite de 10 exécutions.



Graphique 1 : Accélération et efficacité sur 10 exécutions du programme (LOG645, 2017, S.P.)

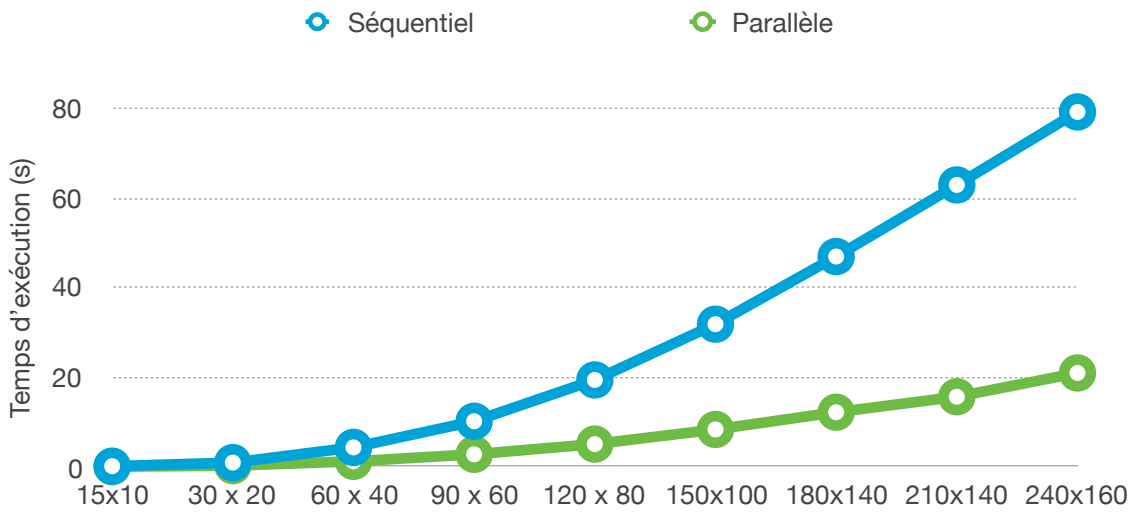
Les accélérations sont peu variables. La variance est sans doute due au fait que d'autres processus tournent sur le serveur distant et occupe du temps processeur. Les résultats des températures sont identiques à chaque exécution, notre algorithme n'utilisant pas la méthode *SOR* ou *Jacobi*, ces méthodes utilisant plutôt une convergence (*finite difference*) qu'une discrétisation de la formule d'Euler. L'algorithme que nous avons développé garantit que les résultats seront toujours calculés de la même façon à chaque exécution.

Courbe du temps en fonction de la taille du problème

Tableau 5 : Temps d'exécution en fonction du nombre de subdivisions sur 4 CPU (LOG645, 2017, S.P.)

Type d'exec.	15x10	30 x 20	60 x 40	90 x 60	120 x 80	150x100	180x140	210x140	240x160
Seq.	0.1923	0.9879	4.3772	10.2135	19.4249	31.8565	46.9463	59.5538	79.1920
Par.	0.0704	0.3102	1.2695	2.8367	5.1153	8.3764	12.2203	15.6730	20.9767

Graphique 2 : Temps d'exécution en fonction du nombre de subdivisions (LOG645, 2017, S.P.)

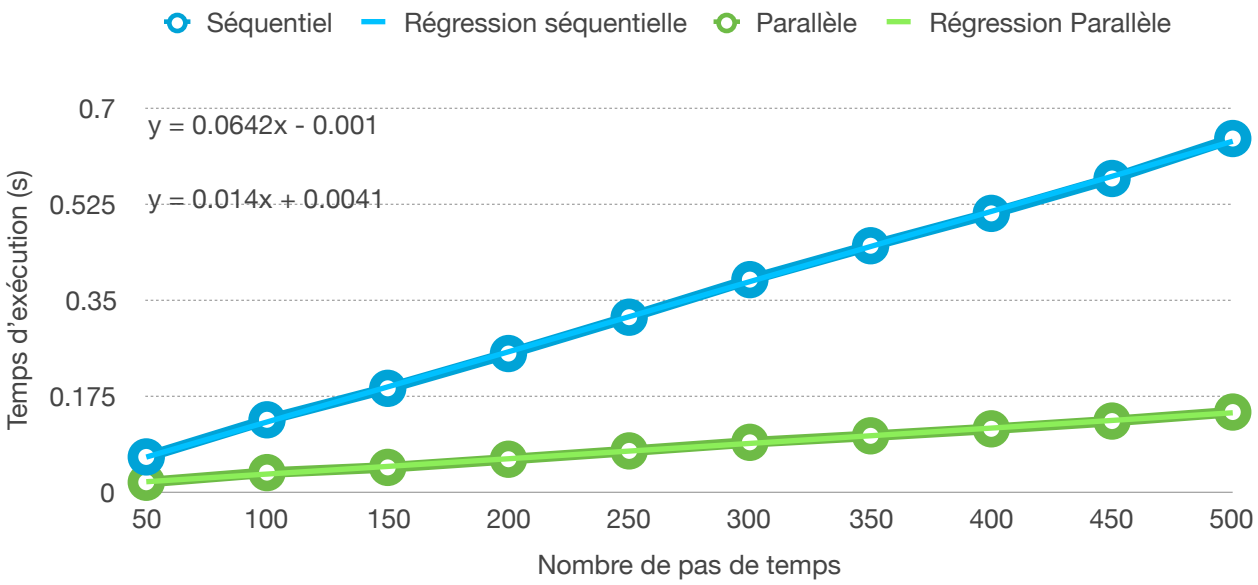


Comme on peut le remarquer, les deux types d'exécutions se distancent très vite l'une de l'autre lors de l'expérience. Plus le domaine devient grand, plus les performances de l'algorithme parallèle augmentent par rapport à l'exécution séquentielle.

Tableau 6 : Temps d'exécution en fonction du nombre de pas de temps sur 4 CPU (LOG645, 2017, S.P.)

Type d'exec.	50	100	150	200	250	300	350	400	450	500
Seq.	0.0635	0.1314	0.1890	0.2526	0.3191	0.3878	0.4498	0.5090	0.5728	0.6459
Par.	0.0172	0.0348	0.0444	0.0593	0.0745	0.0902	0.1021	0.1149	0.1291	0.1456

Graphique 3 : Temps d'exécution en fonction du nombre de pas de temps (LOG645, 2017, S.P.)



Dans ce cas-ci, l'algorithme parallèle prend le devant déjà dans la première exécution avec un pas de temps de 50. L'écart ne cesse de se creuser entre les deux types d'exécutions. Certes, le domaine utilisé ici était très petit (une matrice de 15x10), mais de telles performances aussi constantes augurent très bien pour une utilisation de l'algorithme dans un domaine plus élevé, ce que nous allons étudier dans la prochaine section.

Extrapolation du temps d'exécution en fonction de la taille du problème

Tableau 7 : Temps d'exécution en fonction de M (nombre de colonnes) sur 16 CPU (LOG645, 2017, S.P.)

Type d'exec.	50x50	50x100	50x150	50x200	50x250	50x300	50x350	50x400	50x450	50x500
Seq.	14.017	29.667	44.772	57.523	80.369	100.591	115.913	124.419	133.789	148.934
Par.	1.214	2.525	3.802	5.055	6.145	7.517	8.447	10.203	11.404	12.670

Graphique 4 : Temps d'exécution en fonction de M (nombre de colonnes) (LOG645, 2017, S.P.)

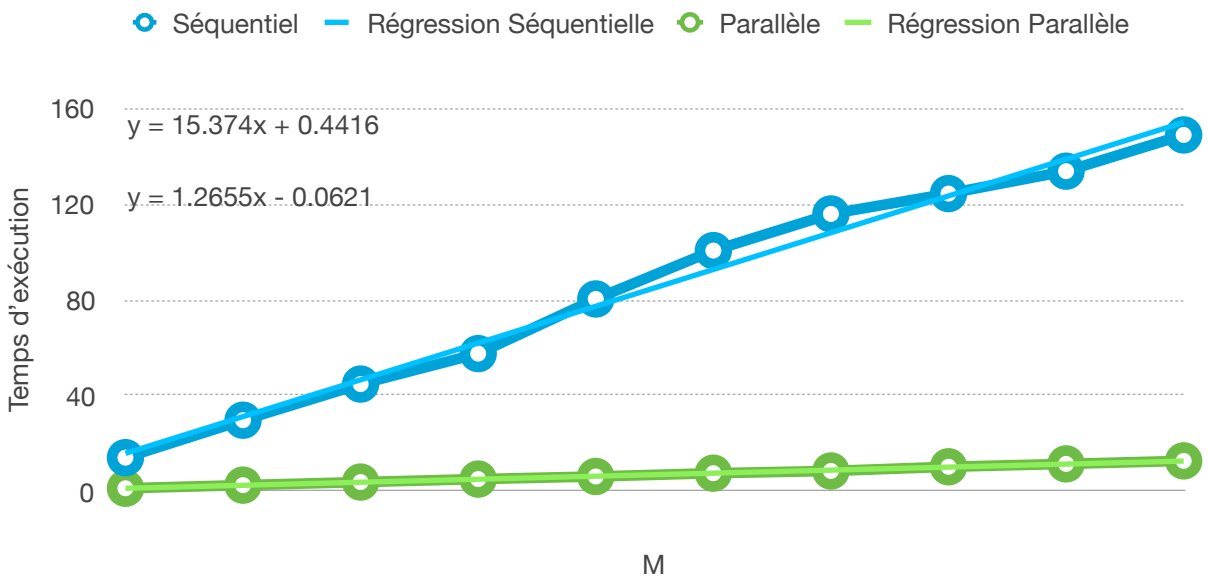
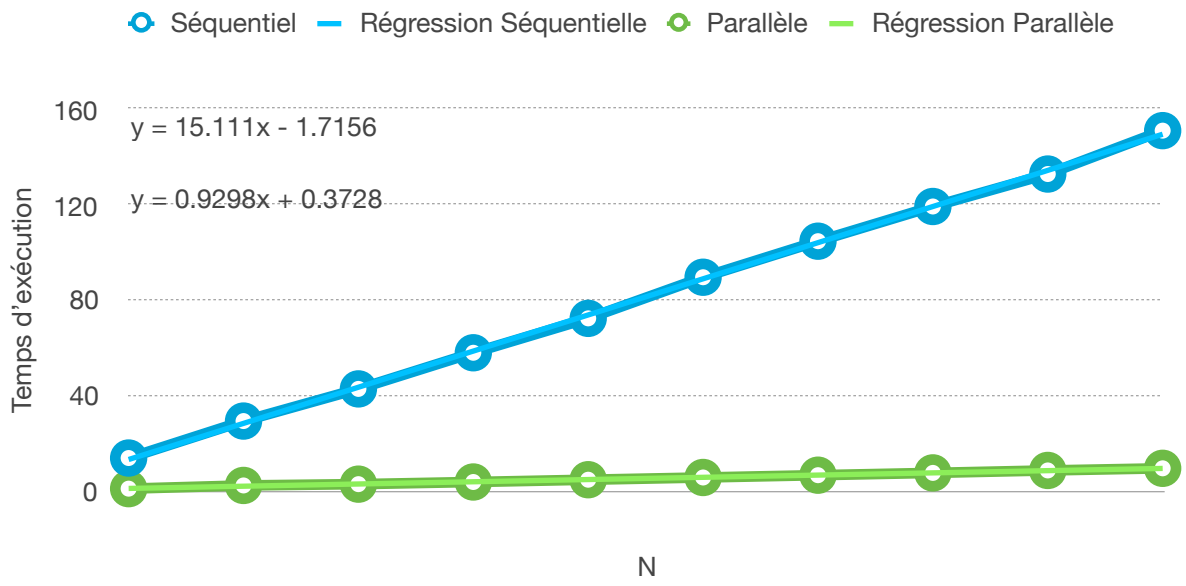


Tableau 8 : Temps d'exécution en fonction de N sur 16 CPU (nombre de lignes) (LOG645, 2017, S.P.)

Type d'exec.	50x50	100x50	150x50	200x50	250x50	300x50	350x50	400x50	450x50	500x50
Seq.	14.0170	29.4724	42.9186	58.0263	72.4113	89.605	104.68	119.18	132.75	150.90
Par.	1.2137	2.5532	3.0943	3.9951	4.9477	5.8807	6.8419	7.8009	8.8247	9.7153

Graphique 5 : Temps d'exécution lors d'une variation de N (nombre de lignes) (LOG645, 2017, S.P.)



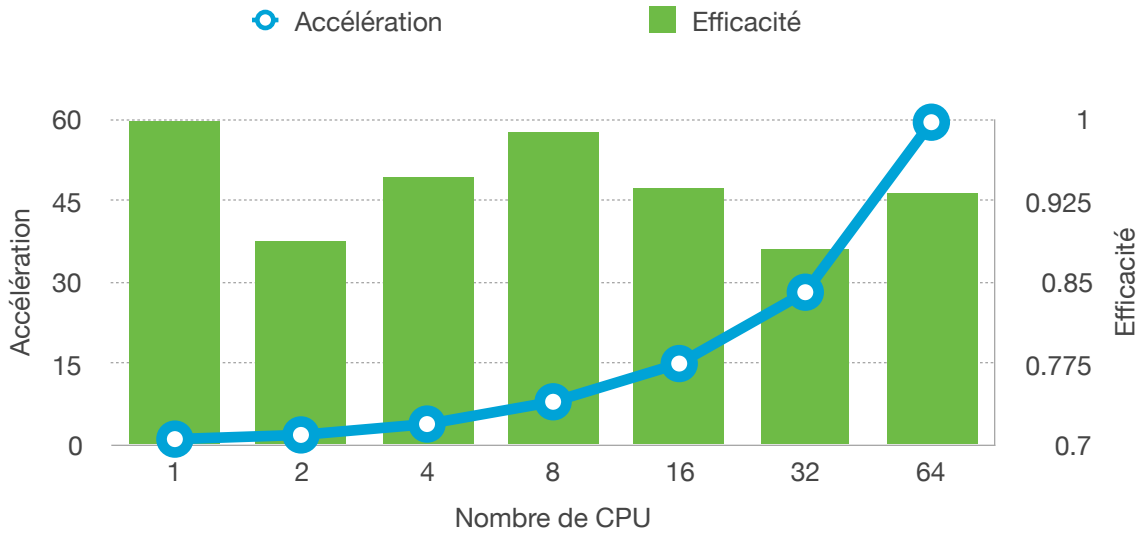
Comme nous pouvons le voir, peu importe si on augmente la taille de la matrice de façon verticale ou horizontale, le temps d'exécution est d'une différence impressionnante entre l'algorithme séquentiel et parallèle. L'extensibilité (*scalability*) de notre algorithme parallèle est excessivement bonne, suivant une pente linéaire extrêmement petite au fur et à mesure que la taille du problème augmente. Le nombre de pas de temps tend à avoir un impact plus important toutefois (graphique 3), mais celle-ci reste toutefois linéaire. Dans le premier cas, où le nombre de colonnes varie, le nombre de communications reste le même, mais la taille de celle-ci augmente, puisqu'une communication envoie toutes les données d'une ligne au processeur voisin. Dans le second cas, où le nombre de lignes varie, le nombre de communications n'augmente pas non plus puisque le nombre de processeurs est stable. Ce qui augmente, toutefois, est le nombre de lignes que doit traiter un CPU de manière séquentielle (*inner matrix rows*). Les communications sont petites, puisque la matrice ne compte que 50 colonnes de largeur. La charge est remarquablement bien répartie et équitable entre les CPU, ce qui permet d'avoir un traitement d'autant plus efficace.

Accélération en variant le nombre de processeurs

Tableau 9 : Accélération en fonction du nombre de processeurs avec une matrice 300x200 sur 200 itérations (LOG645, 2017, S.P.)

Type d'exec.	1	2	4	8	16	32	64
Accélération	1.0000	1.7764	3.7851	7.9072	14.9802	28.1806	59.6387
Efficacité	1.0000	0.8882	0.9463	0.9884	0.9363	0.8806	0.9319

Graphique 6 : Accélération et efficacité en fonction du nombre de CPU (LOG645, 2017, S.P.)



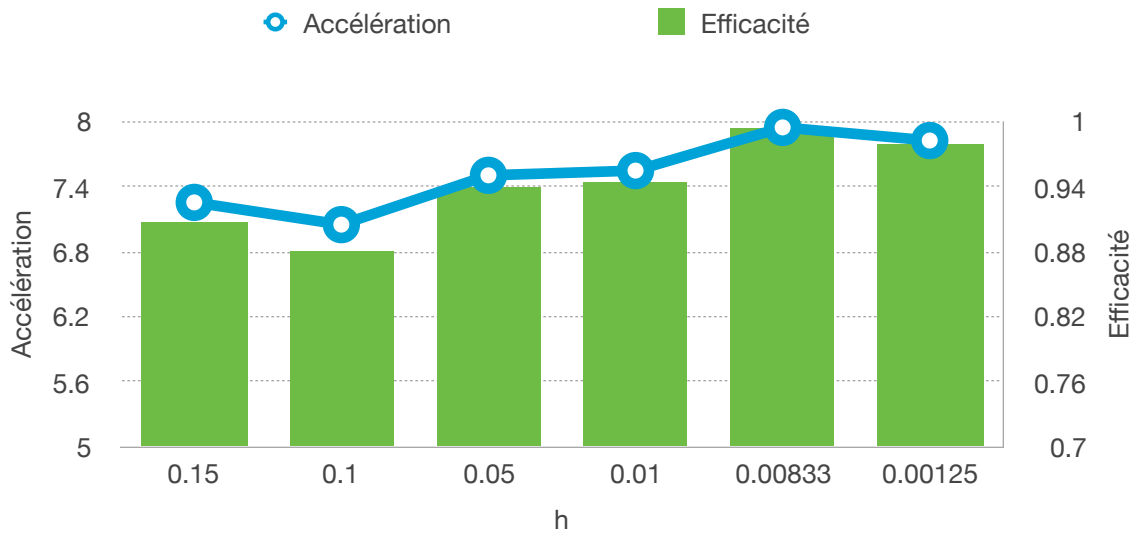
On peut dire que nous n'avons rien de moins qu'une courbe parfaite d'accélération. En fait, l'algorithme, même avec 64 processeurs, n'a pas encore atteint le point culminant où l'accélération stagne, ce qui est pour les deux membres de l'équipe du jamais vu. Normalement, on devrait voir une courbe ascendante suivie d'un plateau, ce dernier étant atteint suite à une trop grande surcharge par rapport aux communications. Or, avec le patron de décomposition 1D, il n'en est pas encore le cas avec 64 processeurs. De plus, l'efficacité reste quand même très acceptable même avec ce nombre élevé de processeurs. La courbe a même l'air d'avoir une allure exponentielle et tout indique que l'utilisation de plus de processeurs pourrait augmenter davantage la rapidité d'exécution de l'algorithme.

Variation de la taille du problème avec le paramètre h

Tableau 10 : Accélération en fonction de la taille de subdivision *h* sur une matrice 100x100 sur 100 itérations avec 8 CPU (LOG645, 2017, S.P.)

Type d'exec.	0.15	0.1	0.05	0.01	0.00833	0.00125
Accélération	7.2581	7.0524	7.5086	7.5524	7.9519	7.8305
Efficacité	0.9073	0.8816	0.9386	0.9441	0.9940	0.9788

Graphique 7 : Temps d'exécution lors d'une variation de taille de subdivision *h* (LOG645, 2017, S.P.)



Avec l'analyse des résultats obtenus et présentés ci-dessus, nous croyons qu'il a été très payant de travailler sur le patron de décomposition géométrique 1 D. Il s'avère que ce patron est excessivement efficace et profite énormément de la présence de plusieurs processeurs. La taille influence de façon très minime le temps d'exécution, rendant ce patron parfaitement extensible avec la taille du problème. L'accélération est également stable en fonction de la taille *h* du problème, comme le montre le graphique 7. Les communications, étant minimes, maximisent l'efficacité et réduisent la surcharge du parallélisme au maximum.

Conclusion

À la lumière des sections précédentes, il est clair que l'algorithme parallèle réalisé par les membres de l'équipe a amélioré grandement l'efficacité de traitement du problème. Le principal objectif qui était de concevoir et d'implémenter une solution parallèle afin d'augmenter les performances de calcul est alors atteint. En effet, alors que l'algorithme séquentiel a un temps d'exécution de 735 secondes dans le temps le plus grand enregistré, l'algorithme parallèle s'exécute en seulement 13 secondes sur 64 processeurs. Une accélération supérieure à 55, ce qui démontre clairement que la répartition sur plusieurs processeurs est optimale pour la simulation de transfert de chaleur et que le patron de décomposition géométrique est plus qu'approprié à ce problème. De plus, rien n'indique que le nombre maximal de processeurs dont nous disposions ait fait atteindre une quelconque limite à l'algorithme.

Lors de la réalisation de l'algorithme parallèle, l'équipe a effectué un processus de conception très exhaustif : deux solutions ont été trouvées. La première fut de distribuer une ligne et une colonne à chaque processus afin de briser les dépendances de flots. Finalement, le temps gagné était minime. L'équipe a plutôt utilisé le patron de décomposition géométrique qui permet de répartir les lignes équitablement entre les processeurs. La communication entre les processus requiert seulement le transfert d'une ligne grâce à la notion de «*ghosting*». La matrice finale est assemblée par le processeur maître. De plus, l'implémentation présentée dans ce rapport pourrait encore faire preuve d'amélioration puisque des communications dites «bloquantes» ont été utilisées. Plusieurs ouvrages suggèrent plutôt d'utiliser des communications non bloquantes comme *MPI_Irecv()* et *MPI_Isend()*, ce qui réduirait encore plus le coût en temps des communications et améliorerait encore plus l'accélération. Toutefois, le manque de temps et d'autres priorités ont fait en sorte que l'équipe s'est contentée de l'implémentation avec des communications standards bloquantes. Toutefois, cette modification pourrait être faite dans une itération future.

En conclusion, la simulation du transfert de chaleur est un problème qui bénéficie de la vitesse de traitement qu'offre le parallélisme. Bien qu'utiliser énormément comme exemple typique de problème ayant plusieurs dépendances, il n'en demeure pas moins complexe. Les membres de l'équipe ont été en mesure d'affiner leur connaissance de la librairie MPI une dernière fois dans le cadre du cours de calculs parallèles. Il aurait été intéressant de développer plusieurs algorithmes et de les soumettre à plusieurs tests afin d'obtenir le plus efficace en fonction de la dimension du problème et du nombre de processeurs. Un arbre de décision du meilleur algorithme aurait alors pu être créé, et le problème serait toujours résolu dans un temps optimal.

Bibliographie

Mattson, Timothy G., Sanders, Beverly A. et Berna L. Massingill. 2004. *Patterns for Parallel Programming,* 1ère édition. Boston : Pearson Education, Inc. 355 p.

Pacheco, Peter S. 1997. *Parallel Programming with MPI*. 1ère édition. San Francisco : Morgan Kaufmann Publishers. 418 p.

Gropp, William, Lusk, Ewing et Rajeev Thakur. 1999. *Using MPI : Portable Parallel Programming with the Message-Passing Interface.* 2e édition. Cambridge : The MIT Press
371 p.

Annexe

Annexe 1 : principale partie du code séquentiel

Ligne	Code
191	`  // Sequential calulation.`
192	`for ( k = 1; k <= timeStep; k++ ) {`
193	`    for ( i = 1; i < lineNumber - 1 ; i++ ) {`
194	`        for ( j = 1; j < columnNumber - 1; j++) {`
195	`            usleep(WAITING_TIME);`
196	`            matrixUkp1[i][j] =`
197	`             ( 1 - 4 * ( discreteTime / ( subDivisionSize * subDivisionSize ) ) ) *`
198	`               matrixUk[i][j] +`
199	`               ( discreteTime / ( subDivisionSize * subDivisionSize ) ) *`
200	`               ( matrixUk[i-1][j] +`
201	`                 matrixUk[i+1][j] +`
202	`                 matrixUk[i][j-1] +`
203	`                 matrixUk[i][j+1]`
204	`               );`
205	`            }`
206	`        }`
207	`// "Copy" the matrixUkp1 to matrixUk by swapping pointers.`
208	`    matrixTemp = matrixUkp1; matrixUkp1 = matrixUk; matrixUk = matrixTemp;`
209	`}`

Annexe 2 : décomposition géométrique 1D

Ligne	Code
556	`void MPE_Decomp1d(int global_num_y, int num_procs, int myid, int *start_y, int *end_y)`
	`{`
557	`    int local_num_y, deficit;`
559	`    local_num_y  = global_num_y / num_procs;`
560	`    *start_y = myid * local_num_y;`
561	`    deficit = global_num_y % num_procs;`
563	`    if(myid < deficit) {`
564	`        *start_y += myid;`
565	`        local_num_y++;`
566	`    }`
568	`    else {`
569	`        *start_y += deficit;`
570	`    }`
572	`    *end_y = *start_y + local_num_y - 1;`
574	`    if (*end_y > global_num_y || myid == num_procs - 1) {`
575	`        *end_y = global_num_y - 1;`
576	`    }`
578	`    // If the number of CPUs is too high, the domain cannot be decomposed. This will return the following error and close all processes.`
579	`    if ( *end_y  - *start_y <= 0 ) {`
581	`        if ( processRank == 0 ) {`
582	`            printf("Cannot partition the domain for parallel execution. Too many CPUs for the vertical size of the domain. Program will now terminate.\n");`
584	`            exit(EXIT_FAILURE);`
585	`        }`
587	`        else {`
588	`            exit(EXIT_FAILURE);`
589	`        }`
590	`    }`
591	`}`