23/05/2017

Projet SFPN

Le plus grand rectangle

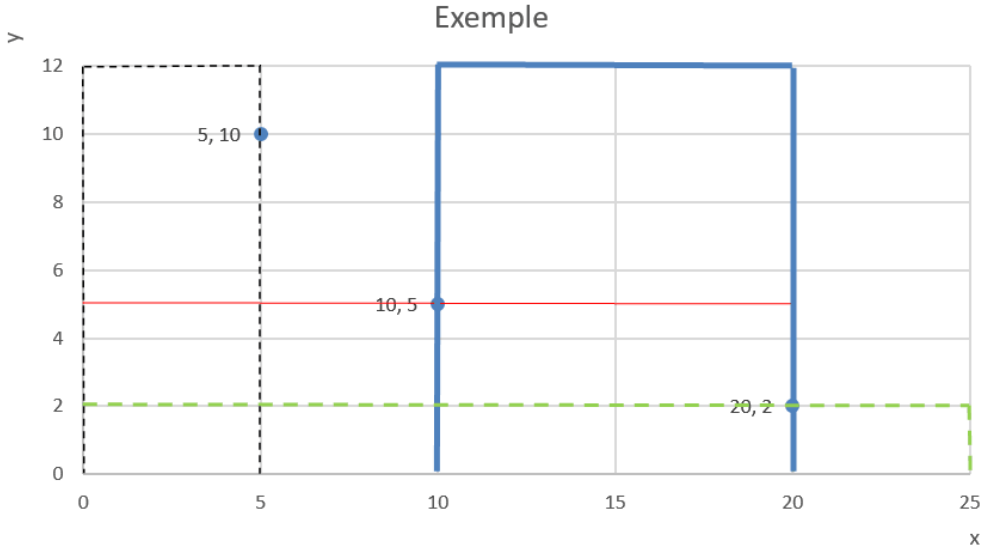
M1 INFO SFPN 2016-2017
SUN Chengcheng
ZHOU Su
Encadrant : Pierre Fortin

Sommaire

I. Introduction

Ce projet aborde un problème d'algorithmique : le calcul du plus grand rectangle à l'intérieur d'un nuage de points (en anglais nous pouvons dire " the largest empty rectangle"), le nombre de points N pouvant atteindre plusieurs millions.Ce problème, qu'on rencontre par exemple dans des concours de programmation (séquentielle), permet en effet d'aborder différents algorithmes, plus ou moins efficaces en séquentiel. Or ces différents algorithmes présentent un « potentiel » de parallélisation très variable. Le but de ce projet est donc d'implémenter des versions séquentielles puis parallèles de ces algorithmes, sur multi-coeur et multi-threads, afin de les comparer et de déterminer si et comment l'impact du parallélisme modifie le classement (en termes de temps de calcul) de ces divers algorithmes[7].

Avec ces différents points dans le système des coordonnées, nous souhaitons dessiner un rectangle dont la base est sur l'axe des x, dont l'intérieur ne contienne aucun des n points et qui soit de surface maximale. Par exemple, l = 25, h = 12, nous avons 3 points (5,10), (10,5), (20,2), nous avons quelques exemples ici : (0,0)(0,12)(5,12)(5,0) de surface 60, (0,5)(20,5)(0,0)(20,0) de surface 100, (10,0)(10,12)(20,12)(20,0) de surface 120, (0,0)(0,2)(25,2)(25,0) de surface 50 … La surface maximale ici est 120.



Nous avons bien réalisé les 4 algorithmes séquentiels en langage C et nous les avons classé par leurs complexités, nous les avons nommé les algorithmes n3, n2, LR (Left-Right), diviser-pour-régner pour les algorithmes de complexité $O(n^3)$, $O(n^2)$, $O(n^2)$(dans le pire cas), O(nlogn). Nous avons démontré leurs complexités et nous avons analysé leurs performances. Après la comparaison entre les algorithmes, nous avons utilisé les méthodes de distribuer les threads pour les tâches sur OpenMP (Open Multi-Processing) qui est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée, et

puis nous avons choisi l'algorithme n2 pour bien adapter MPI (Message Passing Interface) qui est une norme définissant une bibliothèque de fonctions, en fin nous avons souhaité de joindre la méthode OpenMP et MPI pour améliorer la performance.

Dans ce rapport, nous avons présenté les 3 parties principales : les algorithmes séquentiels, les algorithmes avec openMP, les algorithmes avec MPI. Nous avons noté des résultats de l'exécution et dessiné des tableaux pour expliquer la comparaison de la performance, nous avons réfléchi et résumé des résultats par ce projet, nous avons absorbé beaucoup d'expériences dans ce projet.

II. Cadre séquentiel

2.1 L'algorithme de complexité O(n³)

C'est un algorithme naïf qui va chercher les deux points des bornes et la hauteur du rectangle en parcourant tous les points.

2.1.1 L'algorithme

L'algorithme de complexité O(n³)

Entrée : table des points table_points, nombre des points n
Sortie : la surface maximale
Initialiser max_surface ← 0, surface ← 0, min_y ← table_points[n].y
Pour tous les points i et tous les points j après ce point
On parcourir tous les points pour trouver la plus basse hauteur entre i et j et on calcule la surface entre i et j avec cette hauteur minimale
FinPour
On compare les surfaces et trouve la surface maximale
Renvoyer max_surface

2.1.2 La complexité

Pour calculer les surfaces, chaque fois nous avons le temps O(1) en utilisant les 3 points ce que nous avons trouvé, donc nous avons la relation de la complexité :

$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} O(1) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} O(n) = \sum_{i=0}^{n-1} O(n^2) = O(n^3)$

Donc nous avons obtenu la complexité est $O(n^3)$.

2.2 L'algorithme de complexité O(n²)

C'est un algorithme qui va parcourir tous les points pour trouver les deux points des bornes du rectangle avec la hauteur dynamique.

2.2.1 L'algorithme

L'algorithme de complexité $O(n^2)$

Entrée : table des points table_points, nombre des points n
Sortie : la surface maximale
Initialiser max_surface ← 0, surface ← 0
Pour tous les points i
    Soit la plus basse hauteur h est la hauteur du point i+1
    Pour tous les points j après le point i
        On met à jour h = min (h, hauteur j), on utilise h pour calculer la surface entre j+1 et i
        Si i et j sont adjacents
            On calcule la surface entre ces deux points avec la hauteur maximale
        FinSi
    FinPour
FinPour
On compare les surfaces et trouve la surface maximale
Renvoyer max_surface

2.2.2 La complexité

Pour calculer les surfaces, chaque fois nous avons passé le temps de O(1) en utilisant les 2 points ce que nous avons trouvé avec la plus basse hauteur, donc nous avons utilisé la formule de la complexité :

$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(1) = \sum_{i=0}^{n-1}(n - i - 1) = \sum_{j=0}^{n-1} j = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$

Donc la complexité est $O(n^2)$.

2.3 L'algorithme de Left-Right

C'est un algorithme qui va chercher les deux points des bornes du rectangle dynamique : si nous rencontrons le point qui est satisfait à la condition, la boucle va s'arrêter, elle va continuer à calculer les autres.

2.3.1 L'algorithme

L'algorithme de Left-Right

Entrée : table des points table_points, nombre des points n
Sortie : la surface maximale
Initialiser max_surface ← 0, surface ← 0, min_y ← table_points[n].y
    left ← 0, right ← 0
Pour tous les points i
    Soit la plus basse hauteur est la hauteur de i
    On cherche le premier point gauche qui est strictement plus bas que le point i
    Et on cherche le premier point droite qui est strictement plus bas que le point i
    On calcule la surface entre les points gauche et droite avec la hauteur minimale
    On calcule aussi la surface entre i et i+1 avec la hauteur maximale
Finpour
On compare les surfaces et trouve la surface maximale
Renvoyer max_surface

2.3.2 La complexité

Dans le pire cas :

Tous les points forment en ligne parallèle à l'axe x.

D'abord on a parcouru tous les points et trouvé un point i : O(n).

Ensuite, on passe $\frac{n}{2}$ fois pour trouver le point à gauche (resp. droite) et $\frac{n}{2}$ fois pour trouver le point à droite (resp. gauche) : O(n) = O($\frac{n}{2}$)+O($\frac{n}{2}$).

Donc la complexité est O(n) = O(n) * (O($\frac{n}{2}$) + O($\frac{n}{2}$)) = O(n^2)

2.4 L'algorithme de type diviser-pour-régner

C'est un algorithme utilise la notion de diviser-pour-régner, il va distribuer les points en deux parties et il va fait une fonction récursive jusqu'à trouver la meilleure surface[2].

2.4.1 L'algorithme

L'algorithme de type diviser-pour-régner

Entrée : table des points table_points, nombre des points n
Sortie : la surface maximale
Initialiser max_surface ← 0, surface_left ← 0, surface_right ← 0, left ← 0 et right ← l
Récursif (left,right) :
    Si le nombre de points entre left inclus et right inclus < 3
        On compare les surfaces et trouve la surface maximale
        Renvoyer la surface maximale
    Sinon
        Pour tous les points entre left inclus et right inclus
            On cherche la plus basse hauteur à ce point m (s'il y a plusieurs points de même la plus basse hauteur, nous choisissons le point qui a le plus grand x)
        FinPour
        On calcule la surface entre left et right avec cette hauteur minimale
        surface_left = Récursif (left, m)
        surface_right = Récursif (m, right)
        On compare les surfaces et trouve la surface maximale
        Renvoyer max_surface
    FinSi
FinRécursif

2.4.2 La complexité

2.4.2.1 Master theorem

D'après « Considérons un problème qui peut être résolu par un algorithme récursif de la famille diviser pour régner qui opère comme suit : on partage une instance de taille n en sous-problèmes. Il y a des sous-problèmes et chacun est de taille n/b, on résout ces sous-problèmes puis on recombine les solutions partielles en une solution du problème initial. On peut imaginer cette approche comme la construction d'un arbre, où chaque nœud est un

appel récursif, et les enfants sont les instances appelées. Dans le cas décrit ci-dessus, chaque nœud possède a enfants. Chaque nœud répartit les données entre ses enfants et récupère les résultats partiels pour les synthétiser et obtenir la solution globale. La répartition des données entre enfants et l'intégration des résultats ont bien entendu également un coût qui, pour un problème de taille n, est noté par f(n) [1]».

La complexité en temps des algorithmes de cette nature est exprimé par une relation de récurrence de type :

$T(n) = aT(\frac{n}{b}) + f(n)$

n est la taille du problème ;
a est le nombre des sous-problèmes ;
n/b est la dimension des sous-problèmes ;
f (n) est le temps de la combinaison des sous-problèmes.

On peut développer cette relation, en substituant la définition dans la partie droite de l'équation pour obtenir une expression pour le coût total au cas par cas. Mais une expression comme celle fournie par le «master theorem», permet d'exprimer la complexité asymptotique sans passer par un développement explicite.

On suppose donnée la relation de récurrence de la forme:

$T(n) = aT(\frac{n}{b}) + f(n)$ avec a>1, b>1

Où f est une fonction à valeurs entières positives. Le «master theorem» s'énonce comme suit :

Si $f(n)=O(n^c)$ avec $c < \log_b a$ alors $T(n) = O(n^{\log_b a})$

Si $f(n) = O(n^c(\log(n))^k)$ avec $c= \log_b a$ et une constante k>=0, alors $T(n) = O(n^c(\log(n))^{k+1})$

Si $f(n) = \Omega(n^c)$ avec $c > \log_b a$ et s'il existe une constante k<1 telle que, pour n assez grand, on a $a*f(\frac{n}{b}) <= k*f(n)$ (cette condition est appelée parfois la «condition de régularité»), alors on a:T(n) = O(f(n)).

2.4.2.2 La calculation de complexité

1. Dans le meilleur cas :

A chaque étape, on effectue la moitié du travail restant. La partition coupe la liste en deux morceaux du nombre des points $\frac{n}{2}$ (où n est le nombre des points). Pour analyser la complexité de diviser-pour-régner, nous utilisons la méthode master théorème, on sait que T [n] = a T[n/b] + f (n) :

Pour l'algorithme diviser-pour-régner, on distribue le problème en deux partie, donc ici nous pouvons écrire a = 2, b = 2, f (n) = O(n-1), cela satisfait que le 2-ième cas du énoncé.

Nous pouvons obtenir directement la complexité du meilleur cas est $O(n\log n)$.

2. Dans le pire cas :

Tous les points sont triés par ordre croissant ou décroissant, la partition coupe la liste en un morceau de longueur 1 et un autre de longueur n – 1 (où n est le nombre des points).

Nous pouvons obtenir directement la complexité du meilleur cas est $O(n^2)$.

2.5 La comparaison

2.5.1 Le pire cas pour l'algorithme LR et divier-pour-régner

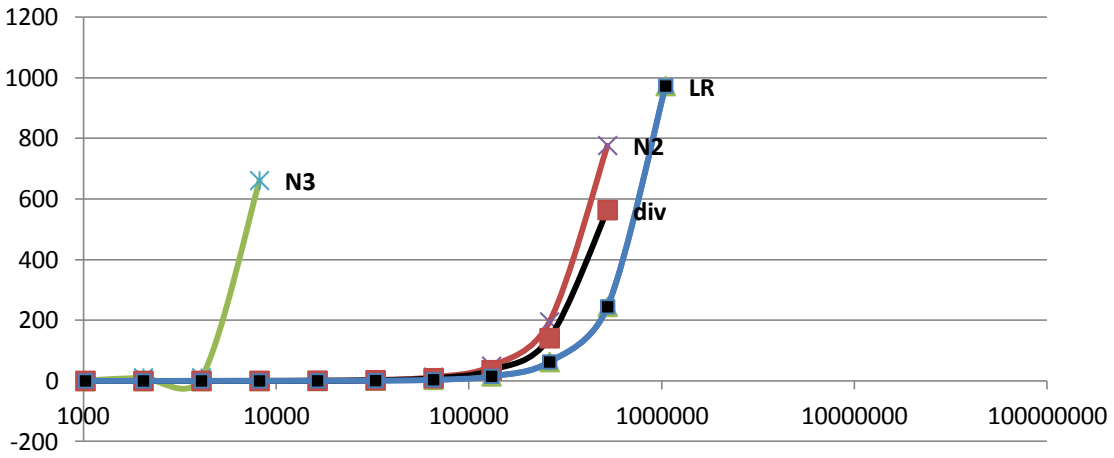


Tableau 2.5.1.1 Tous les points sont triés par ordre croissant

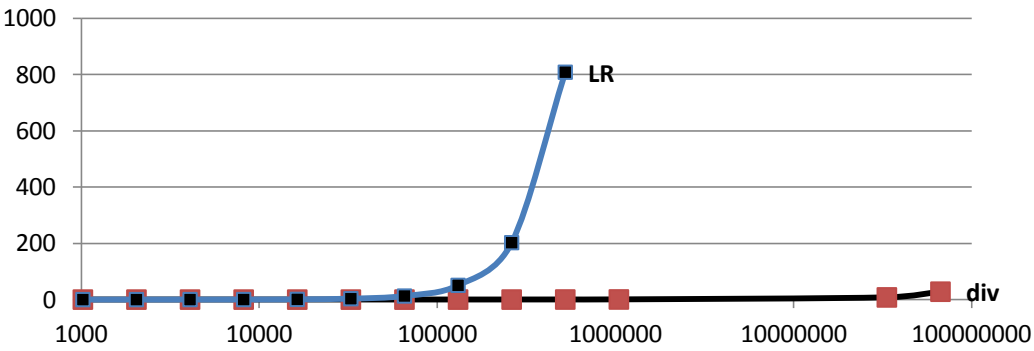


Tableau 2.5.1.2 Tous les points forment en ligne parallèle à l'axe x

Nous avons lancé des algorithmes sur l'environnement de machine PPTI de 4 cœurs et 8 threads, nous pouvons conclure que l'algorithme de diviser-pour-régner est la meilleure solution pour le cas du tableau 2.5.1.2 et l'algorithme de LR est la meilleure solution pour le cas du tableau 2.5.1.1.

2.5.2 Jeu de données aléatoires

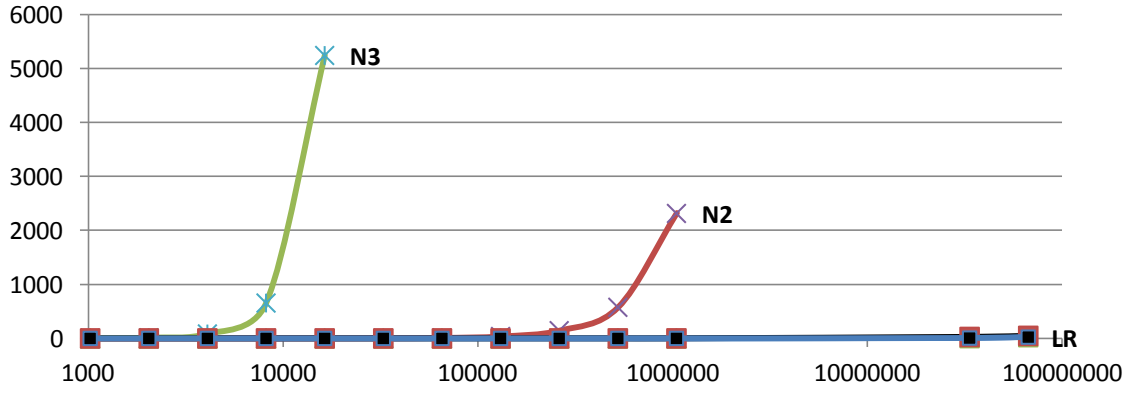


Tableau 2.5.2.1 points générés par ordre croissant de x

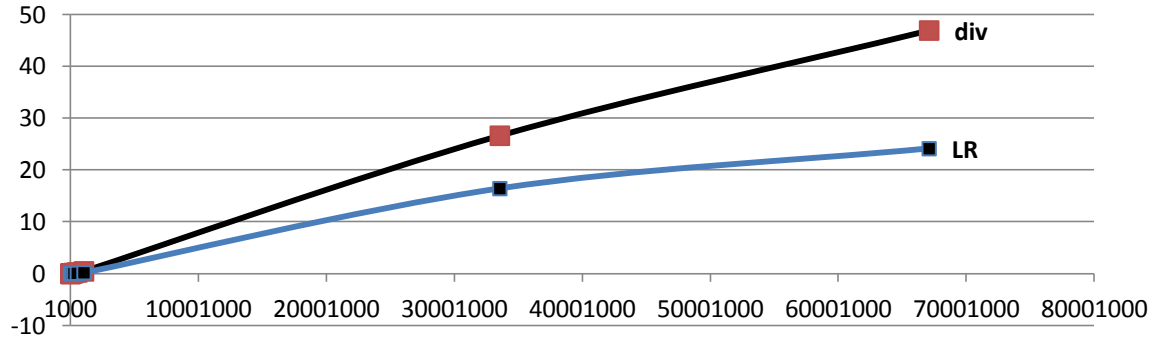


Tableau 2.5.2.2 LR et div points générés par ordre croissant de x

Nous avons lancé des algorithmes sur l'environnement de machine PPTI de 4 cœurs et 8 threads, nous pouvons conclure que l'algorithme de Left-Right est la meilleure solution.

III. Cadre parallel

3.1 OpenMP

OpenMP est une bibliothèque pratique pour nous aider d'augmenter la performance par les threads, le principe de ce méthode est de distribuer les threads, d'utiliser les threads fait exécuter le programme en même temps[4].

3.1.1 L'algorithme de complexité O(n³)

Dans cet algorithme pour la version openMP, nous n'avons pas touché beaucoup sur le code séquentiel et nous n'avons pas changé le cadre de notre algorithme, parce qu'ils ont déjà une directive for pour distribuer les threads associé à l'argument du boucle for. Nous avons ajouté les deux phrases pragma omp parallel et pragma omp for avant le premier boucle for.

L'algorithme de n³ pour openMP

Entrée: table des points table_points, nombre des points n
Sortie: la surface maximale
Initialiser max_surface ← 0, surface ← 0, min_y ← table_points[n].y
#pragma omp parallel shared(max_surface,table_points,n) private(j,surface)
#pragma omp for schedule(dynamic,10) reduction(max:max_surface)
Pour tous les points i et tous les points j après ce point
    On parcourir tous les points pour trouver la plus basse hauteur entre i et j et on
Finpour
On calcule la surface entre i et j avec cette hauteur minimale
On compare les surfaces et trouve la surface maximale
Renvoyer max_surface

3.1.1.1 L'explication des variables

L'option « pragma omp parallel » a commencé à préciser le parallélisme de la partie de l'algorithme principal. Nous avons précisé que max_surface (la surface maximale), table_points (le tableau des points), n (le nombre des points) sont de type partagé car ils doivent être utilisé dans toutes les sessions du parallélisme; j (le compteur du deuxième boucle for) et surface (le surface pour l'instant) sont de type privé puisqu'ils doivent demander une espace de mémoire pour chaque thread pour exécuter le programme correctement sans corrompre la valeur de la variable.

3.1.1.2 L'optimisation

1. schedule(dynamic, chunk)

Pour l'option « pragma omp for », nous avons essayé plusieurs possibilités comme *schedule(static,chunk), schedule(dynamic,chunk), schedule(guided, chunk)*, enfin, nous avons bien comparé et analysé, *schedule(dynamic, chunk)* est le meilleur choix, nous avons essayé en 10, 50, 100 etc. Nous avons choisi la clause « schedule » en mode d'ordonnance dynamique avec chunk_size égal 10, après la comparaison de la performance, *schedule(dynamic, 10),* cela veut dire que le nombre d'itérations va être exécuté par les threads qui vont exécuter 10 d'itérations chaque fois. Quand il finit sa tâche, il obtiendra 10 d'itérations encore jusqu'à la fin de la boucle. Cette option a évité le cas qu'il existe des threads plus rapide et moins rapide, les threads les plus rapide peuvent alors effectuer plus de traitement que les moins rapide, cela va augmenter la performance.

2. réduction (operation : variable)

Premièrement, nous avons essayé d'utiliser critical pour mettre à jour la max_surface, la clause critical a contrôlé un seul thread à la fois pour exécuter la phrase qui est après critical directement. Nous avons essayé d'utiliser la clause reduction pour mettre à jour la max_surface aussi, la reduction peut effectuer une opération que nous avons déjà défini dans la clause reduction, dans cet algorithme, nous avons utilisé reduction (max:max_surface) donc quand les threads veulent stocker et mettre à jour la valeur de max_surface, l'option reduction va comparer le valeur de max_surface automatiquement et enregistrer la plus grande valeur. Après la comparaison entre reduction et critical, la

méthode de reduction est plus rapide que la méthode critical car la méthode de réduction n'a pas besoin d'attendre la terminaison des autres threads, mais la méthode de critical doit attendre à cause de son fonctionnement.

3.1.1.3 Performance

Nous avons testé cet algorithme avec les 2, 4, 8 threads sur la machine PPTI, les 18, 36, 72 threads sur la machine GPU de 2 CPU physiques, chaque CPU a 18 cœurs physiques, 72 cœurs logiques.

temps	1t	2t	4t	8t	18t	36t	72t
1028p	0.83787	0.482311	0.235916	0.182864	0.057685	0.033806	0.044746
2048p	5.31216	2.74547	1.38478	0.646033	0.348407	0.229692	0.18237
4096p	40.9931	22.4607	10.09755	5.15071	2.33544	1.06861	1.05809
8192p	324.132	163.03	85.8108	45.0077	15.6759	9.91898	9.08104
S	1t	2t	4t	8t	18t	36t	72t
1028p	1.0	1.737199	3.551561	4.58193	14.52495	24.7848	18.72507
2048p	1.0	1.934882	3.836104	8.222738	15.247	23.12732	29.12848
4096p	1.0	1.825103	4.059708	7.958728	17.55262	38.36114	38.74255
8192p	1.0	1.988174	3.777287	7.201701	20.67709	32.67796	35.69327
E	1t	2t	4t	8t	18t	36t	72t
1028p	1.0	0.868599	0.88789	0.572741	0.806941	0.688467	0.26007
2048p	1.0	0.967441	0.959026	1.027842	0.847055	0.642426	0.404562
4096p	1.0	0.912552	1.014927	0.994841	0.975146	1.065587	0.538091
8192p	1.0	0.994087	0.944322	0.900213	1.148727	0.907721	0.49574

(temps = temps d'exécution, p = nombre de points, t = nombre de threads, S= accélération, E = efficacité)

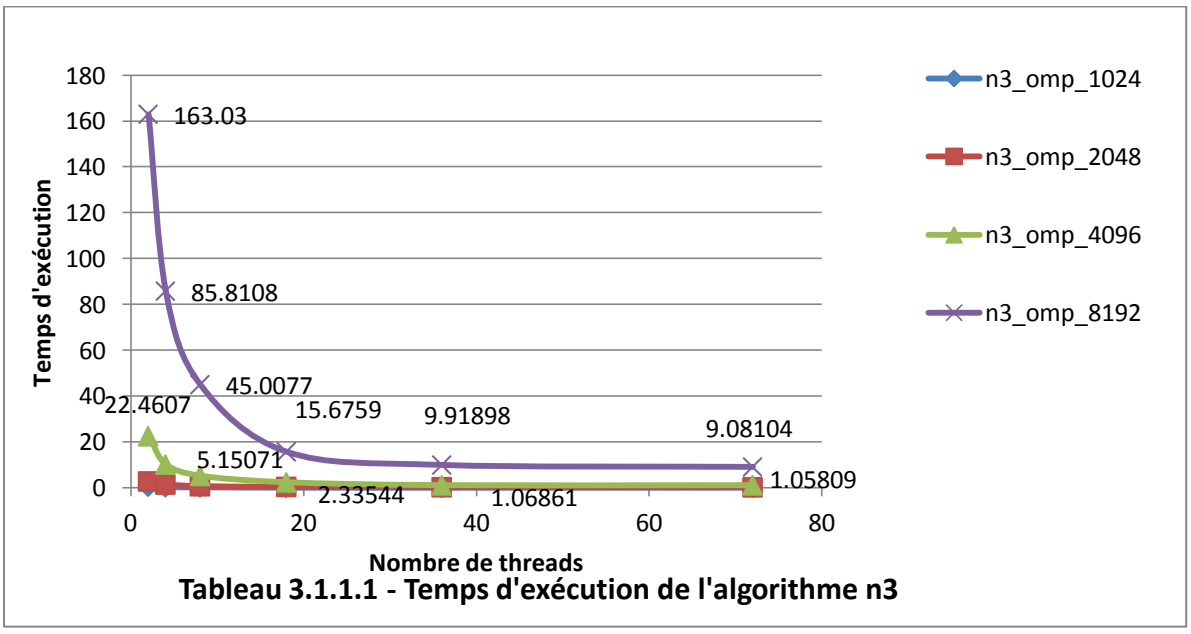


Tableau 3.1.1.1 - Temps d'exécution de l'algorithme n3

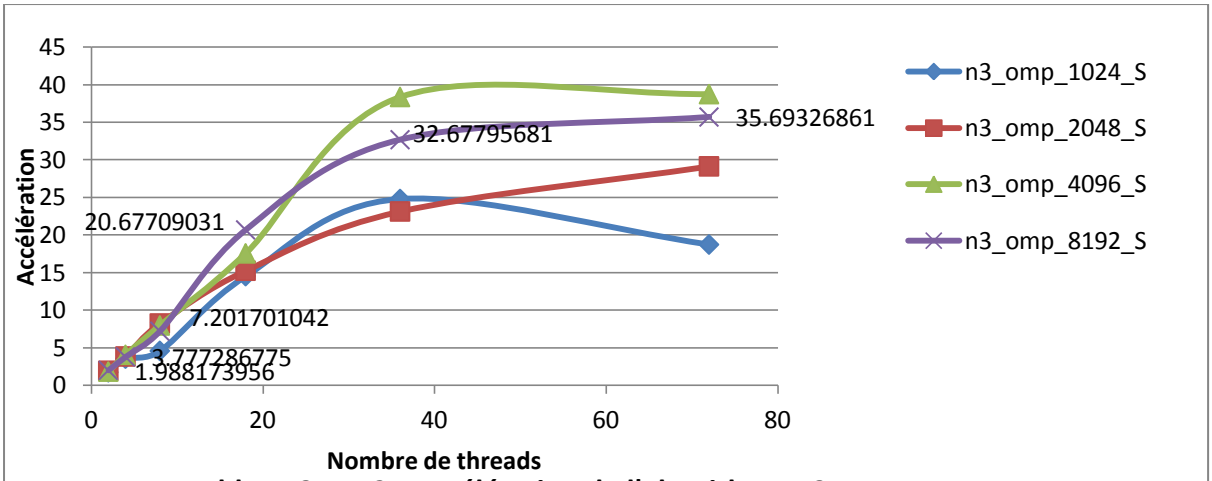


Tableau 3.1.1.2 - Accélération de l'algorithme n3 pour openMP

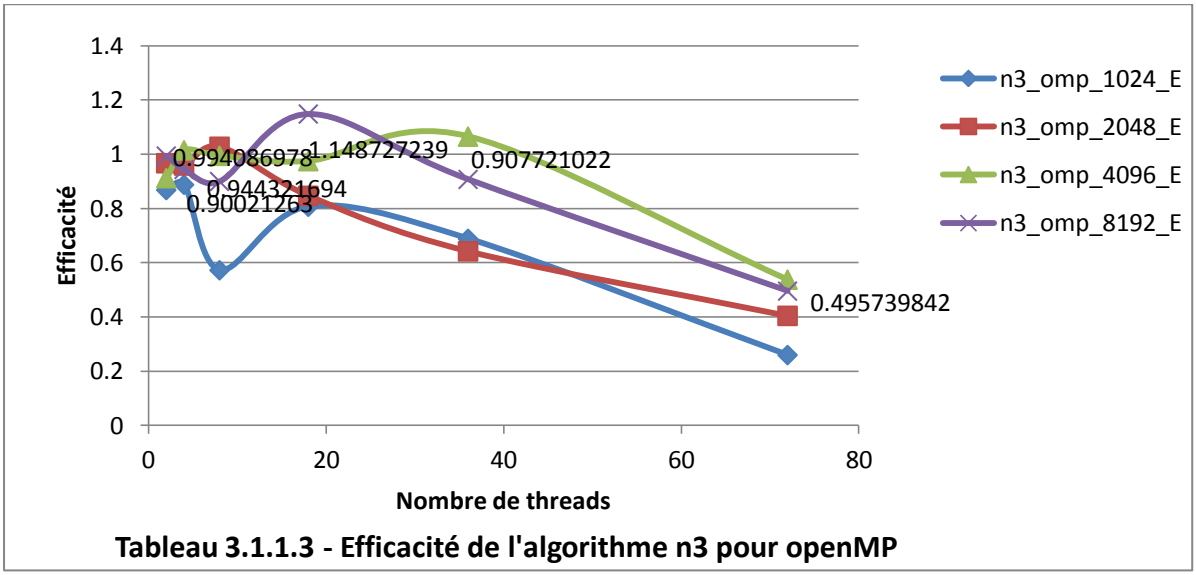


Tableau 3.1.1.3 - Efficacité de l'algorithme n3 pour openMP

Après le calcul de l'accélération et l'efficacité, nous pouvons voir clairement que l'algorithme n3 de version openMP parallélise bien avec une grande accélération et l'efficacité.

3.1.1.4 Conclusion

Pour cet algorithme, nous avons réfléchi aussi d'autre méthode comme « parallel omp section » et « parallel omp task ». Mais cet algorithme n3 est plus naïf que les autres, donc les autres méthodes aurait influencé la performance à cause de distribuer et sauvegarder des espaces libres. Après notre comparaison de la performance, la méthode for est satisfaisante pour cet algorithme.

3.1.2 L'algorithme de complexité $O(n^2)$

Dans cet algorithme pour la version openMP, nous n'avons pas touché beaucoup sur le code séquentiel et nous n'avons pas changé le cadre de notre algorithme, parce qu'ils ont

déjà une directive for pour distribuer les threads associé à l'argument du boucle for. Nous avons ajouté les deux phrases pragma omp parallel et pragma omp for avant le premier boucle for.

L'algorithme de complexité O(**n²**) pour openMP

Entrée: table des points table_points, nombre des points n
Sortie: la surface maximale
Initialiser max_surface ← 0, surface ← 0
#pragma omp parallel shared(max_surface,table_points) private(i,j,surface,min_y,max_surface1)
#pragma omp for schedule(dynamic,10) reduction(max:max_surface)
Pour tous les points i
    Soit la plus basse hauteur h est la hauteur du point i+1
    Pour tous les points j après le point i
        On met à jour h = min (h, hauteur j), on utilise h pour calculer la surface entre j+1 et i
        Si i et j sont adjacents
            On calcule la surface entre ces deux points avec la hauteur maximale
        Finsi
    Finpour
On compare les surfaces et trouve la surface maximale
Renvoyer max_surface

3.1.2.1 L'explication des variables

« pragma omp parallel » a commencé à préciser le parallélisme de la partie de l'algorithme principal. Nous avons précisé que max_surface (la surface maximale), table_points (le tableau des points), n (le nombre des points) sont de type partagé car ils doivent être utilisé dans toutes les sessions du parallélisme ; j (le compteur du deuxième boucle for), surface (le surface pour l'instant), min_y(la hauteur minimale), max_surface1(la surface intermédiaire) sont de type privé puisqu'ils doivent demander une espace de mémoire comme les variable dans l'algorithme n3.

3.1.2.2 L'optimisation

Comme l'algorithme précédent, nous avons bien essayé la clause schedule(static, chunk), schedule(dynamic,chunk), schedule(guided,chunk), la clause critical et réduction, omp parallel for ,omp parallel section, omp parallel task, après nous avons bien comparé toutes les façons et la performance, la clause schedule(dynamic,10) avec reduction(max:max_surface) est la meilleure solution.

3.1.2.3 Performance

Nous avons testé cet algorithme avec les 2, 4, 8 threads sur la machine PPTI, les 18, 36, 72 threads sur la machine GPU de 2 CPU physiques, chaque CPU a 18 cœurs physiques, 72 cœurs logiques.

temps	1t	2t	4t	8t	18t	36t	72t
131072	11.4634	5.77039	2.86874	2.33174	0.782565	0.420715	0.388116
262144	45.4632	18.7434	9.81155	9.3131	2.57461	1.3693	1.25978
524288	181.338	75.0743	39.2351	37.3033	10.1278	5.21716	4.83303
S	1t	2t	4t	8t	18t	36t	72t
131072	1.0	1.98659	3.99597	4.916243	14.6485	27.24742	29.53602
262144	1.0	0.611597	1.168358	1.23089	4.45248	8.371723	9.099525
524288	1.0	2.415447	4.621831	4.861179	17.90497	34.75799	37.52056
E	1t	2t	4t	8t	18t	36t	72t
131072	1.0	0.993295	0.998993	0.61453	0.813805	0.756873	0.410222
262144	1.0	0.305798	0.292089	0.153861	0.24736	0.232548	0.126382
524288	1.0	1.207724	1.155458	0.607647	0.994721	0.9655	0.521119

(temps = temps d'exécution, p = nombre de points, t = nombre de threads, S= accélération, E = efficacité)

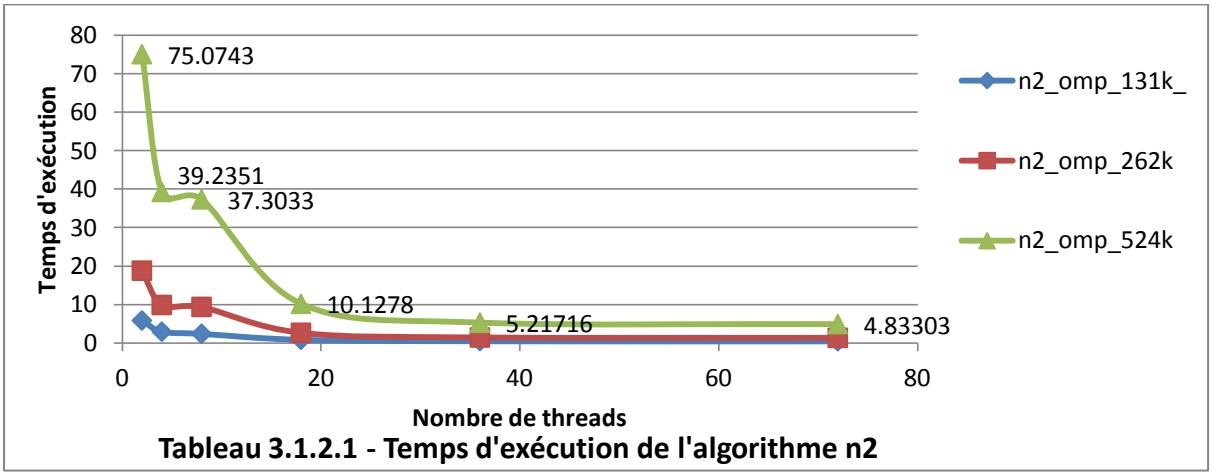


Tableau 3.1.2.1 - Temps d'exécution de l'algorithme n2

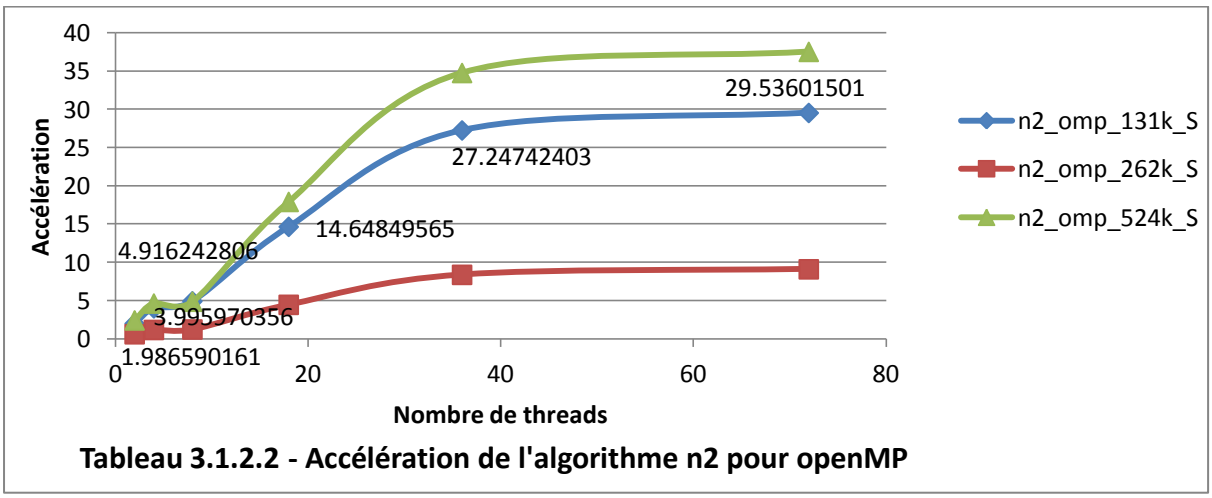


Tableau 3.1.2.2 - Accélération de l'algorithme n2 pour openMP

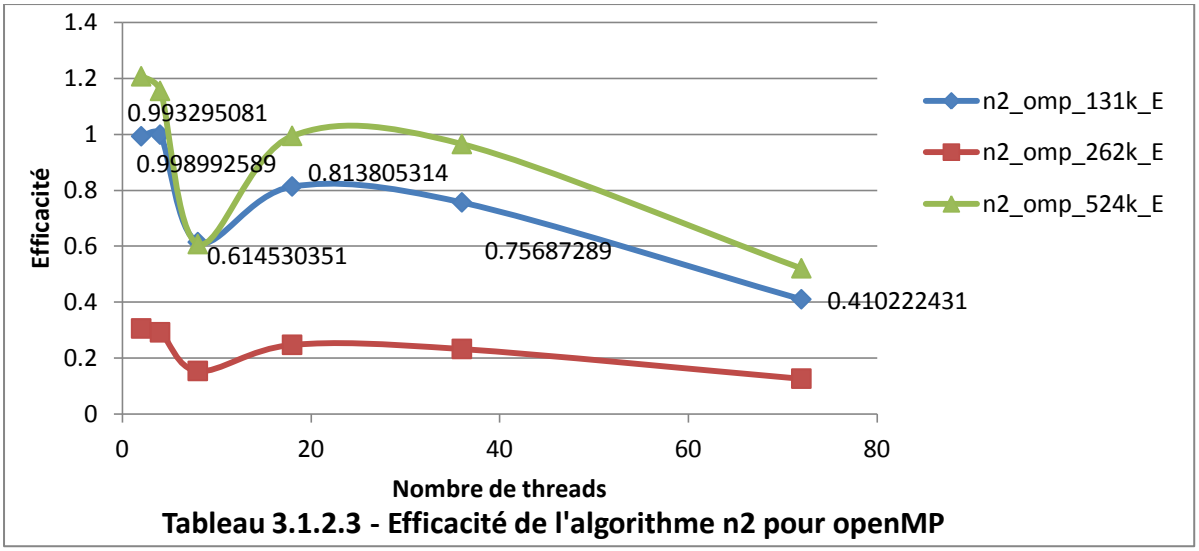


Tableau 3.1.2.3 - Efficacité de l'algorithme n2 pour openMP

Après le calcul de l'accélération et l'efficacité, nous pouvons voir clairement que l'algorithme n2 de version openMP se parallélise bien avec une grande l'accélération et l'efficacité.

3.1.2.4 Conclusion

Cet algorithme n2 est plus naïf que les autres, donc les autres méthodes aurait influencé la performance à cause de distribuer, sauvegarder des espaces libres et ses fonctionnements. Après notre comparaison de la performance, la méthode omp parallel for est satisfaisante pour cet algorithme.

3.1.3 L'algorithme de Left-Right

Dans cet algorithme, c'est aussi comme les 2 algorithmes précédents, nous n'avons pas touché beaucoup sur le code séquentiel et nous n'avons pas changé le cadre de notre algorithme, parce qu'ils ont déjà une directive for pour distribuer les threads associé à l'argument du boucle for. Nous avons seulement ajouté les deux phrases « pragma omp parallel » et « pragma omp for » avant le premier boucle for.

L'algorithme de Left-Right pour openMP

Entrée : table des points table_points, nombre des points n
Sortie : la surface maximale
Initialiser max_surface ← 0, surface ← 0, min_y ← table_points[n].y
left ← 0, right ← 0
**#pragma omp parallel shared(max_surface,table_points) **
private(left,right,surface,surface1,max_surface1,min_y)
#pragma omp for schedule(dynamic,10) reduction(max:max_surface)
Pour tous les points i
Soit la plus basse hauteur est la hauteur de i

On cherche le premier gauche point qui est strictement plus bas que le point i
Et on cherche le premier point droite qui est strictement plus bas que le point i
On calcule la surface entre les points gauche et droite avec la hauteur minimale
On calcule aussi la surface entre i et i+1 avec la hauteur maximale

Finpour

On compare les surfaces et trouve la surface maximale

Renvoyer max_surface

3.1.3.1 L'explication des variables

Nous avons précisé que max_surface (la surface maximale), table_points (le tableau des points), n (le nombre des points) sont de type partagé car ils doivent être utilisé dans toutes les sessions du parallélisme; left (le point ce que nous avons défini la borne gauche de la rectangle), right (le point ce que nous avons défini la borne droite de la rectangle), surface (la surface pour l'instant), min_y(la hauteur minimale), max_surface1(la surface intermédiaire) sont de type privé puisqu'ils doivent demander une espace de mémoire pour initialiser à 0 chaque fois pour chaque thread.

3.1.3.2 L'optimisation

Comme l'algorithme précédent , nous avons bien essayé la clause schedule(static, chunk), schedule(dynamic, chunk), schedule(guided, chunk), la clause critical et réduction, omp parallel for ,omp parallel section, omp parallel task, après nous avons bien comparé toutes les façons et la performance, la clause schedule(dynamic,10) avec reduction(max:max_surface) est la meilleure solution.

3.1.3.3 Performance

Nous avons testé cet algorithme avec les 2, 4, 8 threads sur la machine PPTI, les 18, 36, 72 threads sur la machine GPU de 2 CPU physiques, chaque CPU a 18 cœurs physiques, 72 cœurs logiques.

temps	1t	2t	4t	8t	18t	36t	72t
1048576p	0.463809	0.491142	0.295526	0.292966	0.288178	0.292161	0.264546
33554432p	12.3455	12.3474	7.13811	7.08366	6.95165	6.96692	6.88866
67108864p	24.1419	24.9997	14.1679	13.8172	13.762	13.895	13.5792
E	1t	2t	4t	8t	18t	36t	72t
1048576p	1.0	0.993295	0.998993	0.61453	0.813805	0.756873	0.410222
33554432p	1.0	0.305798	0.292089	0.153861	0.24736	0.232548	0.126382

67108864p	1. 0	1. 207724	1. 155458	0. 607647	0. 994721	0. 9655	0. 521119
S	1t	2t	4t	8t	18t	36t	72t
1048576p	1. 0	0. 944348	1. 569436	1. 58315	1. 609453	1. 587512	1. 753226
33554432p	1. 0	0. 999846	1. 729519	1. 742814	1. 775909	1. 772017	1. 792148
67108864p	1. 0	0. 965688	1. 703986	1. 747235	1. 754244	1. 737452	1. 777859

(temps = temps d'exécution, p = nombre de points, t = nombre de threads, S= accélération, E = efficacité)

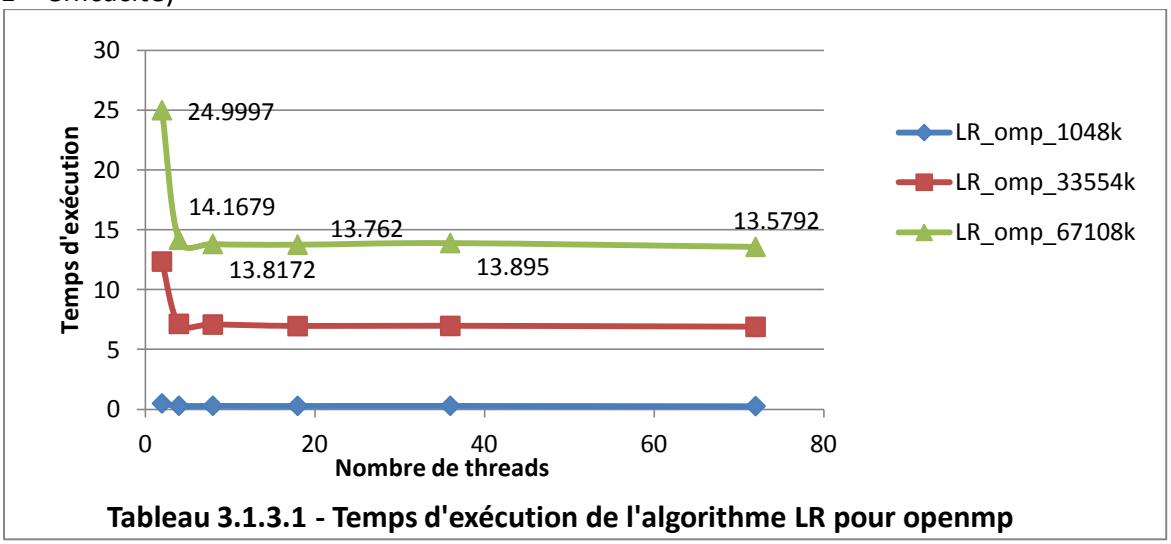


Tableau 3.1.3.1 - Temps d'exécution de l'algorithme LR pour openmp

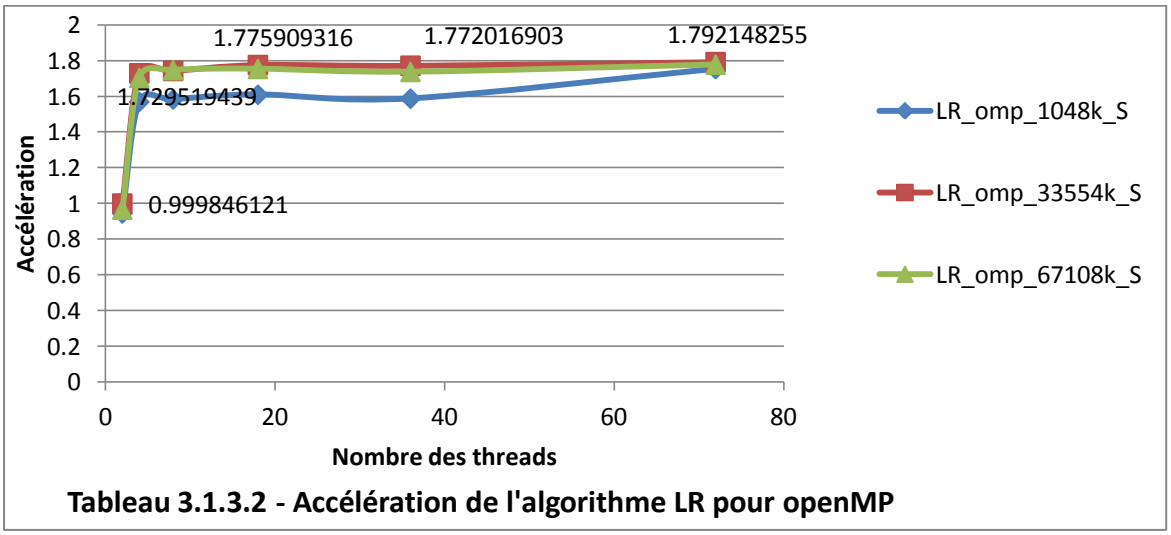


Tableau 3.1.3.2 - Accélération de l'algorithme LR pour openMP

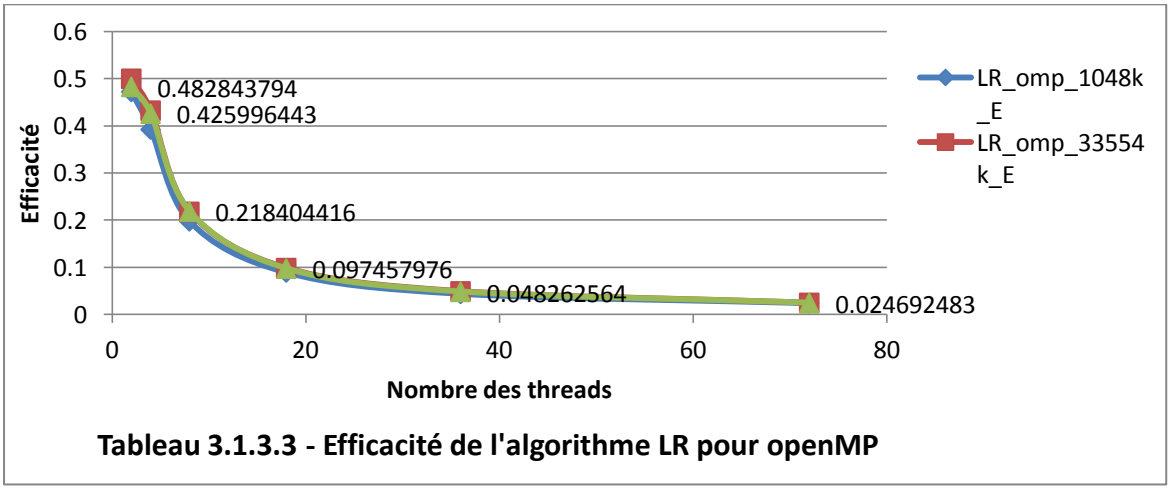


Tableau 3.1.3.3 - Efficacité de l'algorithme LR pour openMP

Après le calcul de temps d'exécution, l'accélération et l'efficacité, nous pouvons voir clairement que l'algorithme Left-Right de version openMP ne parallélise pas bien car l'algorithme séquentiel est vraiment rapide et il a déjà touché le fond du parallélisme de ce niveau.

3.1.3.4 Conclusion

Pour cet algorithme, nous avons réfléchi aussi d'autres méthodes pour ajouter des directives openMP, comme la directive critical qui contrôle un seul thread à la fois pour exécuter le bloc d'une directive critical. Mais cet algorithme Left-Right est plus spécial que les autres, donc les autres méthodes aurait influencé la performance à cause de distribuer et sauvegarder des espaces libres.

Dans cet algorithme, nous avons utilisé la boucle while dans la session de la boucle for, nous avons réfléchi de changer la boucle while à s'adapter la forme openMP, mais cela est impossible puisque le programme cherche les deux bornes du rectangle dynamique, nous ne pouvons pas contrôler la distribution des threads et ces deux bornes trouvé ont des ordres, ces ordres influence le calcul de la surface à la fin, donc nous ne pouvons pas changer l'algorithme sans utiliser while. Après notre comparaison de la performance, la méthode for est satisfait à cet algorithme mais les performances parallèles sont mauvaises.

3.1.4 L'algorithme de type diviser-pour-régner

Dans cet algorithme pour la version openMP, nous avons utilisé la notion de *task*, car il est idéal pour la fonction récursive. Nous avons d'abord ajouté les deux phrases *« #pragma omp parallel »* et « *#pragma omp single »* avant dans la fonction récursive, la directive *parallel* a précisé que la région de parallélisme, la directive *single* a défini une portion de code qui sera exécutée par un seul thread dans une région parallèle[5].

L'algorithme de diviser-pour-régner pour openMP

Entrée: table des points table_points, nombre des points n
Sortie: la surface maximale
#pragma omp parallel
#pragma omp single
Initialiser max_surface ← 0, surface_left ← 0, surface_right ← 0, left ← 0 et right ← l
length ← right - left
Récursive(left, right)
Si length < 8192
    Exécuter le code séquentiel
Sinon
    Pour tous les points entre left inclus et right inclus

On cherche la plus basse hauteur à ce point m (s'il y a plusieurs points de même la plus basse hauteur, nous choisissons le point qui a le plus grand x)

On calcule la surface entre left et right avec cette hauteur minimale

/* Partie openMP */

Si le nombre de points de la partie gauche > 4096

#pragma omp task untied firstprivate(left,m,surface1,surface1_max)

surface_left = Récursif (left, m)

Sinon

surface_left = Récursif (left, m)

FinSi

Si le nombre de points de la partie gauche > 4096

#pragma omp task untied firstprivate(right,m,surface1,surface1_max)

surface_right = Récursif (m, right)

Sinon

surface_right = Récursif (m, right)

FinSi

#pragma omp taskwait

On compare les surfaces et trouve la surface maximale

Renvoyer max_surface

FinPour

Finsi

FinRécursif

3.1.4.1 L'explication des variables

La clause firstprivate(left,m,surface1,surface1_max) peut définir left(le point de la borne gauche du rectangle), m(le point intermédiaire pour séparer les 2 partie gauche et droite), surface1(la surface pour ce niveau), surface1_max(la surface total pour sauvegarder la surface maximale dans tous les niveaux) sont privés et ils ont gardé l'état initial qui sont défini avant de cette session. Après nous avons tombé dans l'appel récursif de la partie gauche. Pour la partie droite c'est presque pareil que la manipulation de la partie gauche, seulement le différent argument droite (le point droite de la borne droite du rectangle).

3.1.4.2 L'optimisation

1. Seuil

Premièrement, nous avons essayé d'utiliser omp task directement, à cause de l'exécution de chaque récursif, le programme va créer une tâche n'importe que la taille de tâche, ce sera perdre beaucoup de temps pour la création des nouvelles tâches et lire-écrire dans la case mémoire. Donc nous avons ajouté un seuil pour contrôler l'exécution des parties séquentielles et openMP en raison du nombre de point. Nous avons testé beaucoup

de chiffre pour définir ce seuil, après la comparaison de la performance, nous avons choisi 8192 est le meilleur cas ce que nous avons trouvé.

Pour ne pas créer des trop petites tâches, dans le corps de l'algorithme de openMP, nous avons ajouté aussi un seuil avant de l'entrée dans l'appel récursif, nous avons contrôlé le nombre de points dans la partie gauche, si inférieur à 4096, nous avons entré directement dans l'appel récursif, sinon, nous avons créé une nouvelle tâche pour l'appel récursif. Pour la partie droite, nous avons ajouté ce seuil aussi. Nous avons essayé beaucoup de chiffres pour définir ce seuil, après la comparaison de la performance, nous avons choisi 4096 est le meilleur cas ce que nous avons trouvé.

Nous avons essayé d'utiliser *task final* et ajouté *if-else* directement, enfin if-else est le meilleur choix.

2. Untied

Nous avons essayé de utiliser la task tied et untied, l'option untied pour contrôler les threads qui choisissent dynamiquement les tâches, ils peuvent choisir les tâches que les autres threads qui les ont créé, les threads peuvent reprendre tous les tâches si elle est suspendue. Mais pour tied, les threads ne peuvent que exécuter les tâche qu'ils ont crées eux-mêmes. Après la comparaison de la performance, nous avons choisi untied qui est le meilleur.

3.1.4.3 Performance

Nous avons testé cet algorithme avec les 2, 4, 8 threads sur la machine PPTI, les 18, 36, 72 threads sur la machine GPU de 2 CPU physiques, chaque CPU a 18 cœurs physiques, 72 cœurs logiques.

temps	1t	2t	4t	8t	18t	36t	72t
1048576p	0.342994	0.360687	0.359536	0.350292	0.354109	0.356527	0.363196
33554432p	26.6458	8.5935	8.74884	8.27107	8.36989	8.22283	8.37896
67108864p	46.9458	17.5831	17.016	16.5327	16.8409	16.0982	16.2736
S	1t	2t	4t	8t	18t	36t	72t
1048576p	1.0	0.950946	0.953991	0.979166	0.968611	0.962042	0.944377
33554432p	1.0	3.100692	3.045638	3.221566	3.18353	3.240466	3.180084
67108864p	1.0	2.669939	2.758921	2.839572	2.787606	2.916214	2.884783
E	1t	2t	4t	8t	18t	36t	72t
1048576p	1.0	0.475473	0.238498	0.122396	0.053812	0.026723	0.013116
33554432p	1.0	1.550346	0.76141	0.402696	0.176863	0.090013	0.044168
67108864p	1.0	1.334969	0.68973	0.354947	0.154867	0.081006	0.040066

(temps = temps d'exécution, p = nombre de points, t = nombre de threads, S= accélération, E = efficacité)

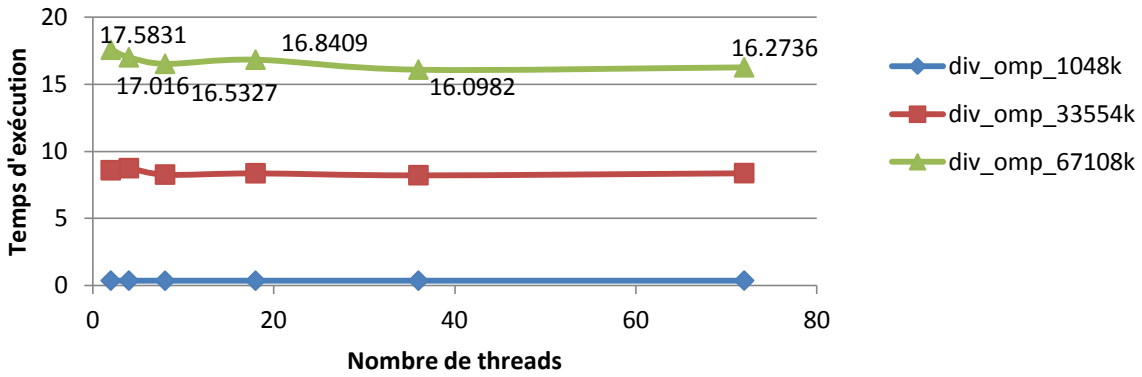


Tableau 3.1.4.1 - Temps d'exécution de l'algorithme Diviser-pour-régner pour openMP

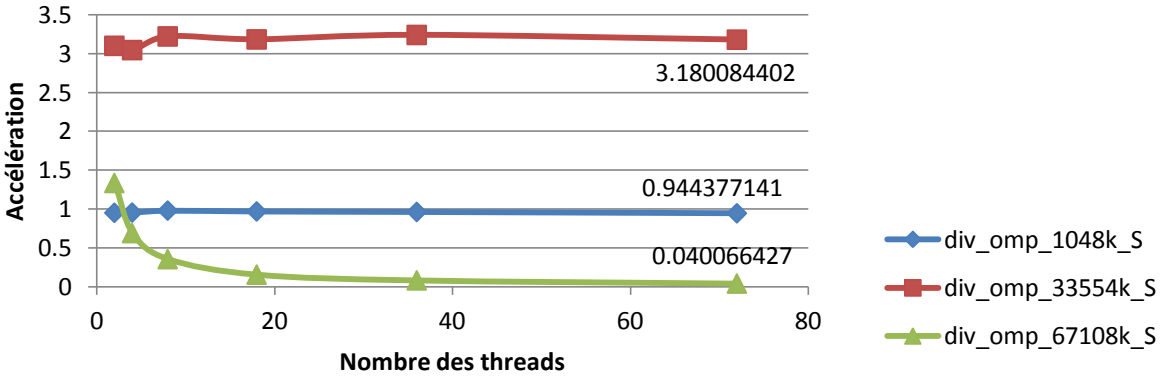


Tableau 3.1.4.2 -Accélération de l'algorithme Diviser-pour-régner pour openMP

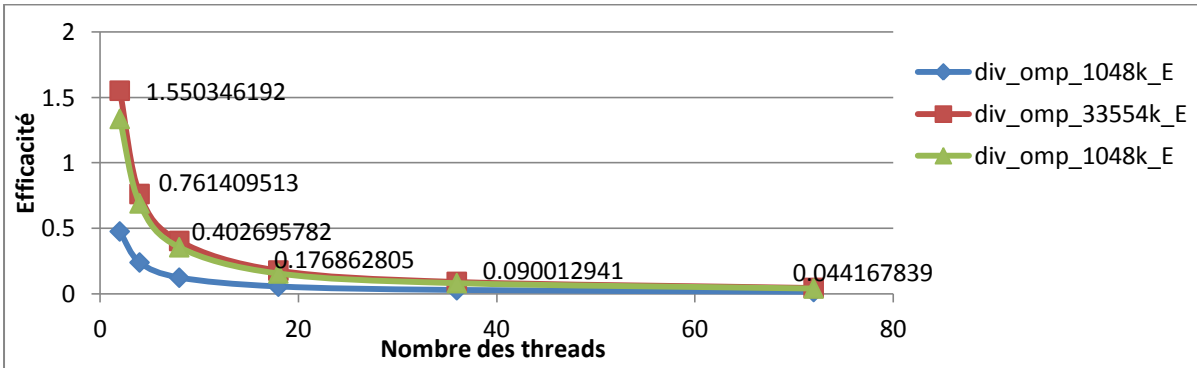


Tableau 3.1.4.3 -Efficacité de l'algorithme Diviser-pour-régner pour openMP

Après le calcul de temps d'exécution, nous pouvons voir clairement que l'algorithme diviser-pour-régner de version openMP ne parallélise pas bien car l'algorithme séquentiel de type diviser-pour-régner est vraiment rapide et il a déjà touché le fond du parallélisme de ce niveau.

3.1.4.4 Conclusion

Pour cet algorithme, nous avons réfléchi aussi d'autre méthode pour ajouter des

directives openMP, comme la directive flush, le programme va bien partager des mémoires, donc cela va influencer la performance, mais après la comparaison de la performance, nous avons choisi task untied avec if-else qui est le meilleur.

3.1.5 Comparaison des différents code du openMP

Le temps d'exécution de n3 et n2 sont très lents, donc ici nous n'avons pas les comparer avec l'algorithme LR et l'algorithme de type diviser-pour-régner en même nombre de points.

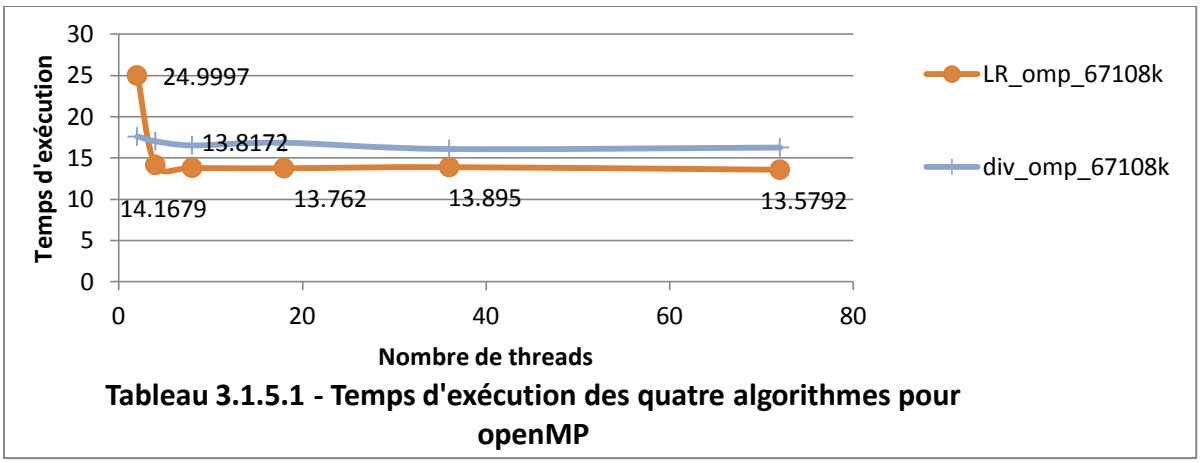


Tableau 3.1.5.1 - Temps d'exécution des quatre algorithmes pour openMP

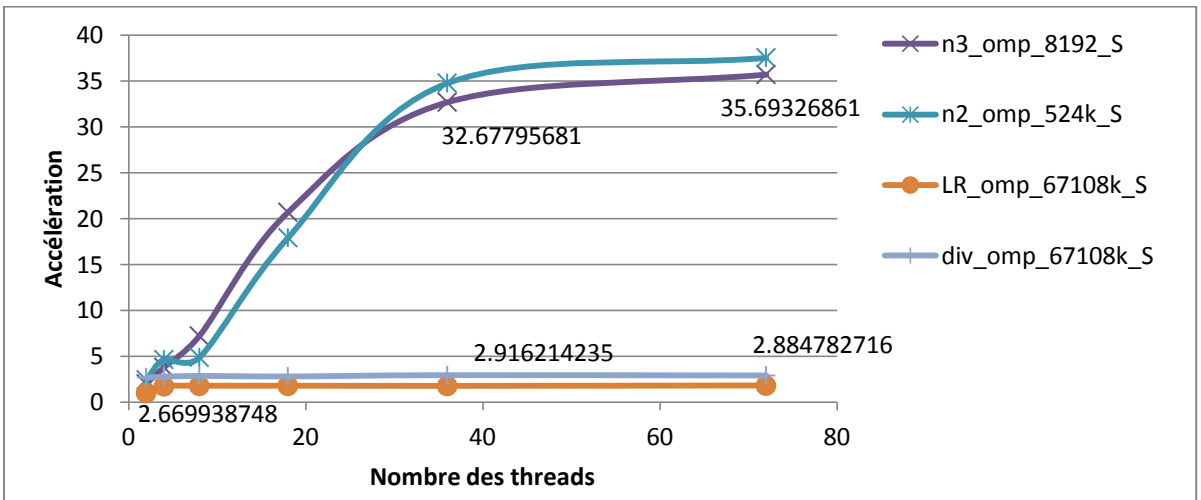


Tableau 3.1.5.2 - Accélération des quatre algorithmes pour openMP

Après la comparaison des quatre algorithmes, quand nombre de points augmente jusqu'à 67108 milles points, le plus rapide algorithme est Left-Right, la plus grande accélération est l'algorithme N2 et N3.

3.2 MPI

Après avoir utilisé openMP, nous utilisons donc la bibliothèque MPI qui permet de réaliser facilement un programme distribué qui peut utiliser plusieurs machines pour les calculs. MPI permet alors de se passer de la gestion de la couche réseaux du programme, et

gère automatiquement la création des processus répartis sur les machines[6].

3.2.1 Parallélisme à équilibrage de charge statique

L'équilibrage de charge statique consiste à chaque processus calcule alors toutes les mêmes quantités d'informations pour la première boucle. Les machines peuvent déterminer quels est le plus grand rectangle vide dont ils doivent faire le calcul, compte tenu du fait qu'ils connaissent la partie de tableau, le nombre de processus existant, leur rang dans l'environnement et la taille de tâche qu'ils doivent calculer. Avec tous ces éléments, ils exécutent le calcul et trouvent la plus grande rectangle dans sa partie et envoient le résultat à maître machine, maître va comparer les surfaces et trouver le plus grande rectangle vide.

Nous avons parallélisé l'algorithme de complexité $O(n^2)$ en utilisant la méthode équilibrage de charge statique. Pour l'algorithme de complexité $O(n^3)$ et l'algorithme de Left-Right, nous pouvons les paralléliser en même façon .

L'algorithme d'équilibrage à charge statique

```
Require: La tableau des points  { espace de travail }
Require: size ≥ 2 {Nombre de processus}
Require: rank > 0 {Rang du processus courant}
Require: root = 0 {Rang du processus maître}
Require: tailleplage > 0 {Taille de calcul: nombre de point diviser par size }
Si rang != size -1
        Pour i de tailleplage x rang  à tailleplage x (rang+1)  faire
                L'algorithme de complexité O(n²)
        FinPour
Sinon
        Pour i de tailleplage x rang  à n
                L'algorithme de complexité O(n²)
        FinPour
Finsi
Si rang !=0
        envoyer(root, résultat )
Sinon
        recevoir(rang, résultat)
        compare les résultat et trouve le plus grande rectangle vide.
Finsi
```

3.2.2 Parallélisme à équilibrage de charge dynamique

Nous avons considéré la méthode l'équilibrage de charge dynamique, qui consiste aux machine maître qui va faire une liste des tâche et distribuer les tâche à l'autre Les processeurs les plus rapide peuvent alors effectuer plus de traitement que les moins rapide,

cela va augmenter la performance de la calculation, mais nous avons utilisé les machines PPTI, nous supposons qu'ils ont la même capacité de calculation, donc nous avons choisi la méthode l'équilibrage à charge statique.

3.2.2.1 Parallélisme à maître –esclave

Maître-esclave est un modèle d'un processus ou un serveur est le maître, l'autre (ou plusieurs d'autres) sont le(s) esclave(s). Le maître donne des ordres à l'esclave qui les exécute, le maître va faire une liste de tâche pour les esclaves qui sont libre. Les esclaves calculent les tâches et renvoyent le résultat au maître.

Ce modèle est bien utilisé à l'algorithme de diviser-pour-régner, la machine maître d'abord exécute l'algorithme pour obtenir un nombre suffisant de noeuds, elle distribue des noeuds aux esclaves. Nous avons bien l'essayé mais nous n'avons pas assez de temps pour l' implémenter complètement.

3.2.3 Performance

Les tests de performances ont été réalisés en lançant l'environnement MPI sur la machine GPU de 2 CPU physiques, chaque CPU a 18 cœurs physiques, 72 cœurs logiques.

temps	1t	2t	4t	8t	18t	36t	72t
131072p	11.4634	8.614751	5.070281	2.65791	1.44267	0.946954	0.51964
262144p	45.4632	34.0038	19.9718	10.6398	5.00936	2.53669	2.0685
524288p	181.338	136.029	79.4663	42.5177	19.6944	12.8906	9.96487
S	1t	2t	4t	8t	18t	36t	72t
131072p	1.0	1.330671	2.2609	4.312938	7.945961	12.10555	22.06027
262144p	1.0	1.337004	2.27637	4.272937	9.07565	17.92225	21.97883
524288p	1.0	1.333083	2.281948	4.265	9.207592	14.06746	18.19773
E	1t	2t	4t	8t	18t	36t	72t
131072p	1.0	0.665336	0.565225	0.539117	0.441442	0.336265	0.306393
262144p	1.0	0.668502	0.569092	0.534117	0.504203	0.49784	0.305261
524288p	1.0	0.666542	0.570487	0.533125	0.511533	0.390763	0.252746

(temps = temps d'exécution, p = nombre de points, t = nombre de threads, S= accélération, E = efficacité)

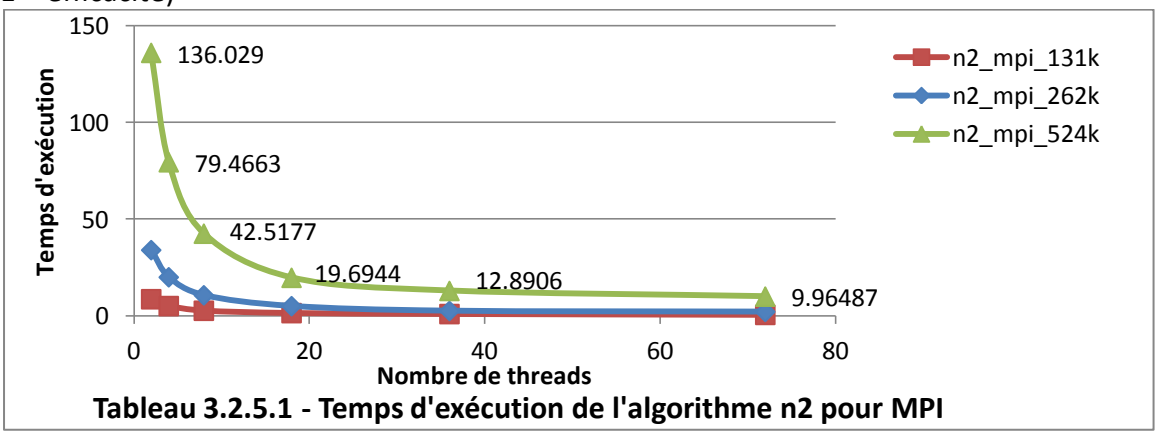


Tableau 3.2.5.1 - Temps d'exécution de l'algorithme n2 pour MPI

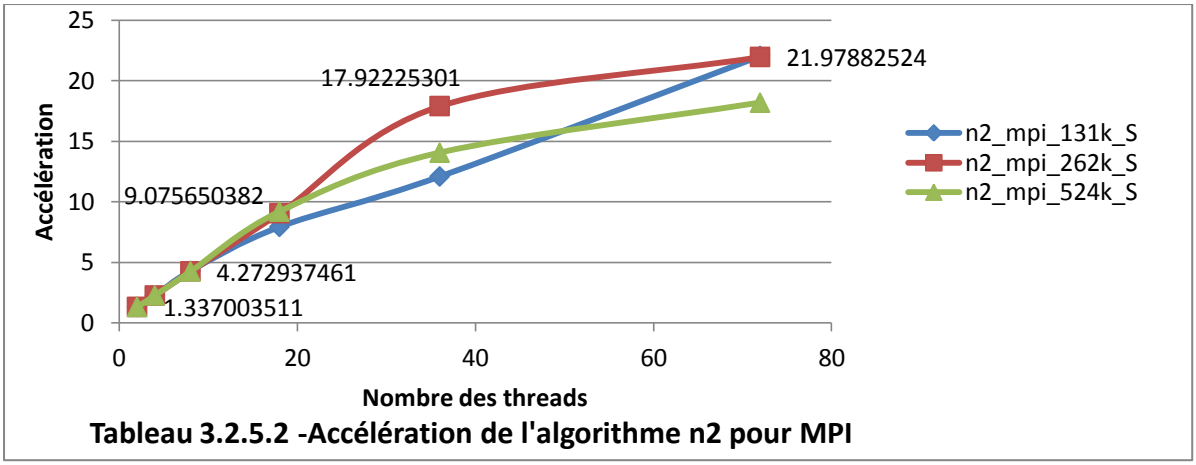


Tableau 3.2.5.2 -Accélération de l'algorithme n2 pour MPI

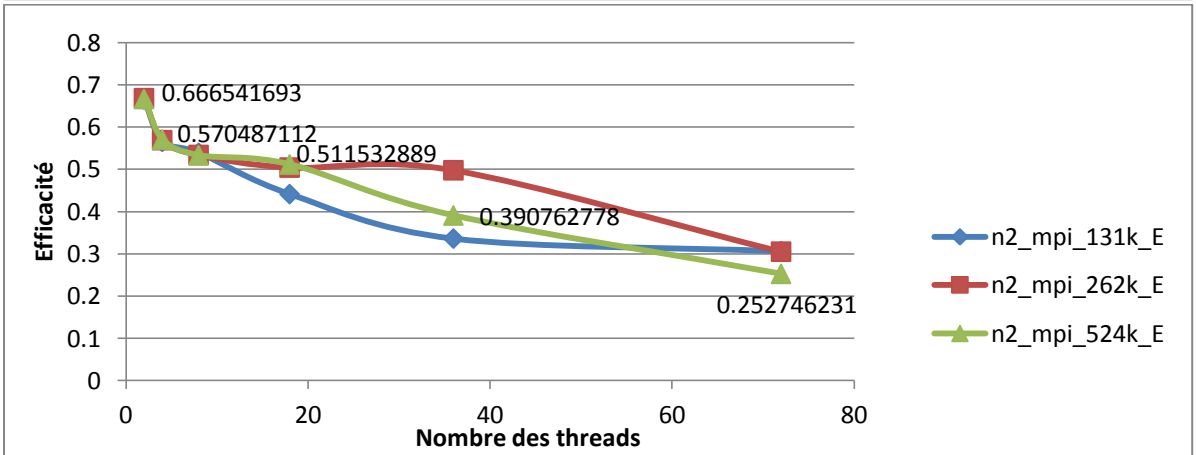


Tableau 3.2.5.3 -Efficacité de l'algorithme n2 pour MPI

Après la calculation de l'accélération et l'efficacité, nous pouvons voir clairement que nous pouvons très bien paralléliser l'algorithme n2 en utilisant la méthode l'équilibrage à charge statique avec MPI. Nous avons une grande l'accélération et l'efficacité.

3.3 MPI + openMP

Après avoir utilisé openMP et MPI, nous avons considéré d'utiliser les deux méthodes en même temps, nous avons essayé d'écrire les codes mais nous n'avons pas bien les tester. Ce sera une bonne utilisation de multi-core avec multi-threads car MPI va augmenter la performance avec les processeurs, openMP va augmenter la performance avec les threads. Dans la base de l'algorithme de MPI, nous pouvons juste ajouter les codes #pragma omp for avant la boucle première for comme nous avons écrit en partie 3.1 c'est-à-dire chaque machine peut exécuter la boucle avec plusieurs threads.

IV. Conclusion

Nous avons bien analysé des algorithmes et les parallélisé en utilisant des différents méthodes, chaque méthode a ses propres caractéristiques, nous les avons comparés au ci-dessous :

4.1 OpenMP

4.1.1 Avantage

1. Espace d'adressage globale plus facile à utiliser;
2. Le partage des données entre les tâches est rapide;
3. Facile à implémenter.

4.1.2 Inconvénient

1. L'utilisateur devra s'assurer que les conflits d'accès à la mémoire sont synchronisés;
2. Problèmes de "Race detection", les outils permettant de les détecter ne sont pas encore matures[8].

4.2 MPI

4.2.1 Avantage

1. Open MPI distribue les processus séquentiellement sur les cœurs disponibles;
2. Librairie complète pour le calcul parallèle[9];
3. Bien utiliser les ressources de calculation.

4.2.2 Inconvénient

1. Difficile à implémenter, il va changer beaucoup d'algorithme séquentiel;
2. Difficile à tester et modifier l'erreur.

4.3 Comparaison

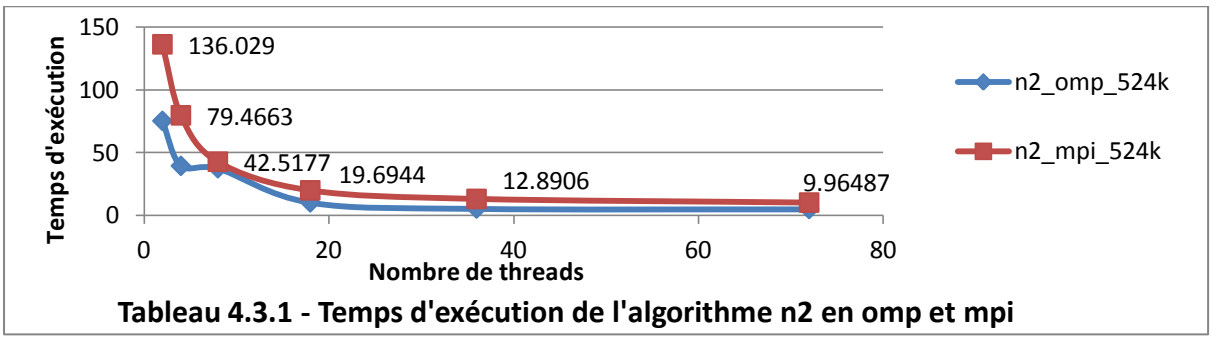


Tableau 4.3.1 - Temps d'exécution de l'algorithme n2 en omp et mpi

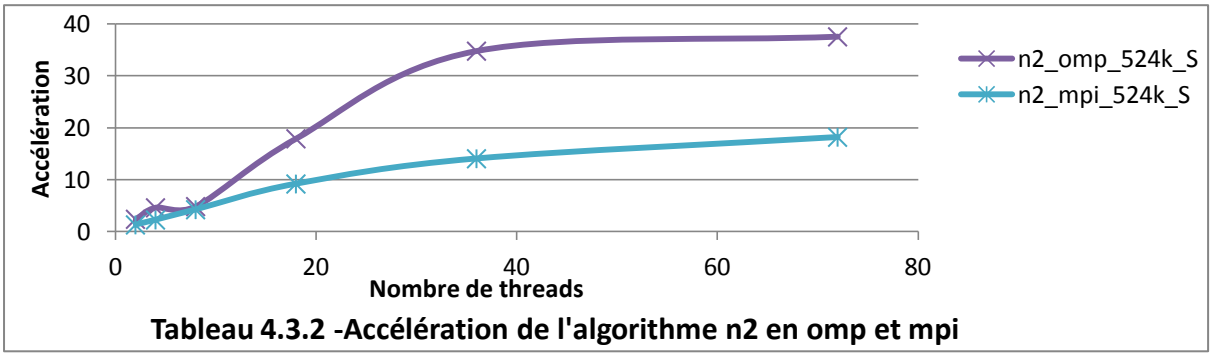


Tableau 4.3.2 -Accélération de l'algorithme n2 en omp et mpi

Après la comparaison les deux algorithmes nous pouvons lister les avantages et les inconvénients des deux méthodes:

1. OpenMP est plus rapide que MPI, la plus grande accélération est en version openMP aussi.

2. OpenMP est limité par le nombre de coeurs disponibles sur le noeud tandis que MPI peut s'exécuter sur plusieurs noeuds.

4.4 OpenMP + MPI

Nous avons consisté à paralléliser les mêmes codes sur plusieurs niveaux, un niveau OpenMP au sein d'un noeud au mémoire partagé et un niveau MPI entre les différents noeuds du calcul[10]. L'intérêt est que nous pouvons augmenter la performance avec les processeurs et les threads. Mais ce sera difficile à implémenter et il faut éviter de beaucoup d'erreurs entre le partage de mémoire sous multi-cœurs.

4.5 Bilan

Après la comparaison de l'exécution du temps entre les algorithmes séquentiels, nous pouvons conclure que l'algorithme LR est la meilleure solution car dans le cas normal, nous avons obtenu le plus petit de temps d'exécution. L'algorithme LR a une structure spéciale grâce à la boucle while qui peut sauter la boucle immédiatement.

Après la comparaison des algorithmes parallèles, nous pouvons conclure que l'algorithme LR a le meilleur temps d'exécution, mais le meilleur effet du parallélisme est l'algorithme n3 et n2. Parce que les algorithmes séquentiels de n3 et n2 sont pire que les autres, le parallélisme les aide à améliorer bien la performance.

Dans ce projet, nous avons bien étudié comment réfléchir et réaliser les algorithmes pour résoudre le problème « le plus grand rectangle », nous avons comprendre les cadres du parallélisme avec les bibliothèques OpenMP et MPI pour des algorithmes. Pour la continuation de ce projet, nous voudrions explorer un algorithme de complexité linéaire et paralléliser cet algorithme.

V. Références

1.https://fr.wikipedia.org/wiki/Master_theorem 2017-04-03.

2.https://openclassrooms.com/courses/algorithmique-pour-l-apprenti-programmeur/une-classe-d-algorithme-non-naifs-diviser-pour-regner,2017-04-03.

3.http://pageperso.lif.univmrs.fr/~michel.vancaneghem/mait/documents/cours3_8.pdf

4.http://www.openmp.org/resources/tutorials-articles/ 2017-03-23.

5.http://www.ecs.umass.edu/ece/andras/pact_2003_tutorials.htm#OpenMP,2017-03-23.

6.http://mpi-forum.org/docs/docs.html 2017-03-23.

7. B Chazelle, RL Drysdale, DT Lee, Computing the Largest Empty Rectangle,SIAM J. SIAM Journal on Computing, 1986 - SIAM

8. Sanjiv Shah, Mark Bull, OpenMP,Proceedings of the 2006 ACM/IEEE conference on Supercomputing,2006-11-11.

9. Snir Marc, MPI : the complete reference, Cambridge, Mass. : MIT Press, c1996

10.B. Lange and P. Fortin ,Parallel dual tree traversal on multi-core and many-core architectures for astrophysical N-body simulations, 2014.