PROJET SFPN

Problème du plus grand rectangle

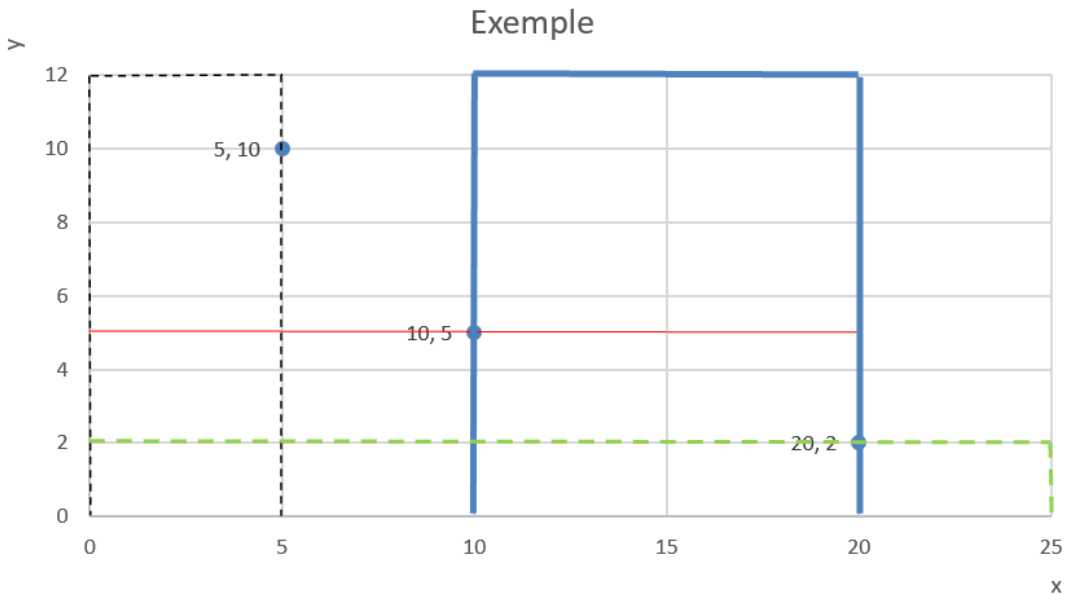
M1 INFO SFPN 2016-2017
Encadrant : Pierre Fortin
SUN Chengcheng
ZHOU Su

UPMC
SORBONNE UNIVERSITÉS

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.

- Le but de ce projet : 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.



II. Séquentiel

- L'algorithme de complexité $O(n^3)$
- L'algorithme de complexité $O(n^2)$
- L'algorithme de Left-Right de complexité $O(n^2)$
- L'algorithme de type diviser-pour-régner de complexité $O(n\log n)$

L'algorithme de complexité $O(n^3)$

- **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

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 mise 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 maximal
- **Renvoyer** max_surface

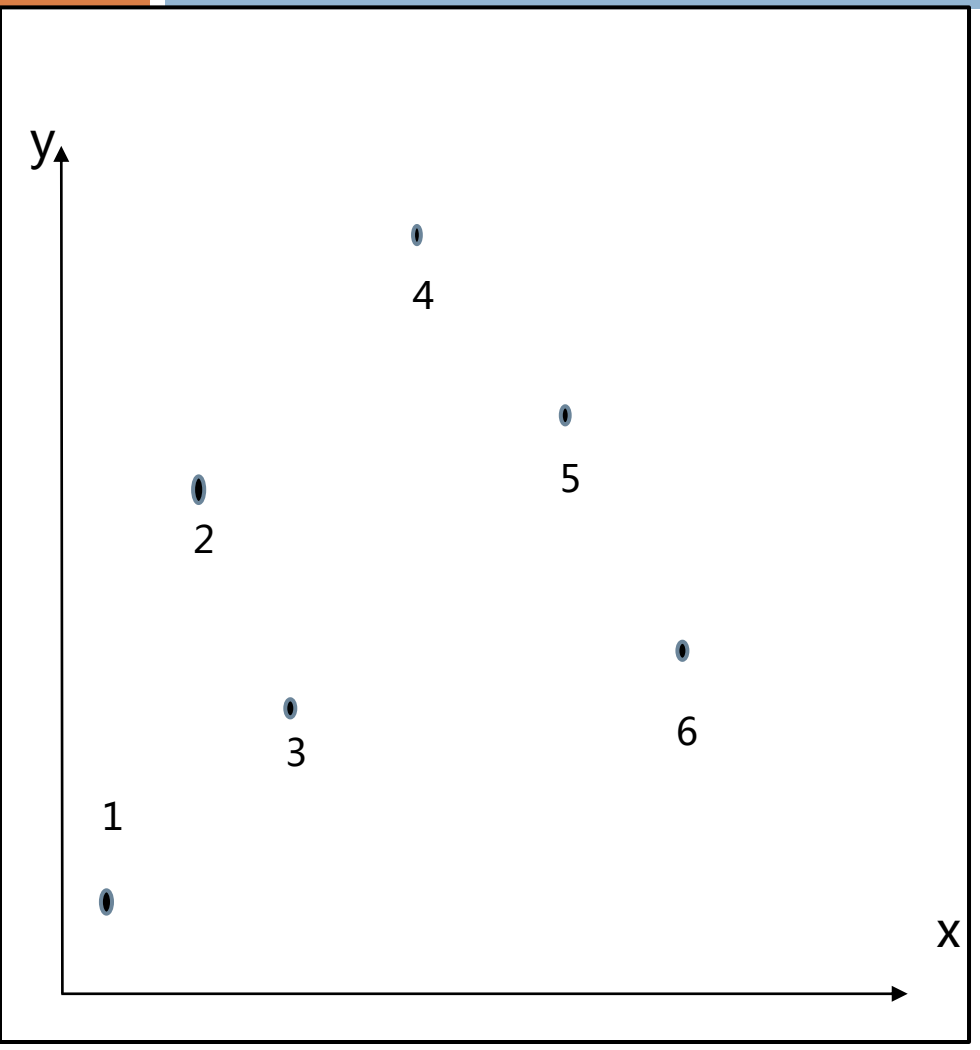
L'algorithme de Left-Right de complexité $O(n^2)$

- **Entrée :** table des points table_points, nombre des points n
- **Sortie :** la surface maximale
- **Initialiser** max_surface ← 0, surface ← 0, 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 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 maximal
- **Renvoyer** max_surface

L'algorithme de type diviser-pour-régner de complexité O(nlogn)

- **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)
- **Finpou**
- 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**

Jeu de données aléatoires



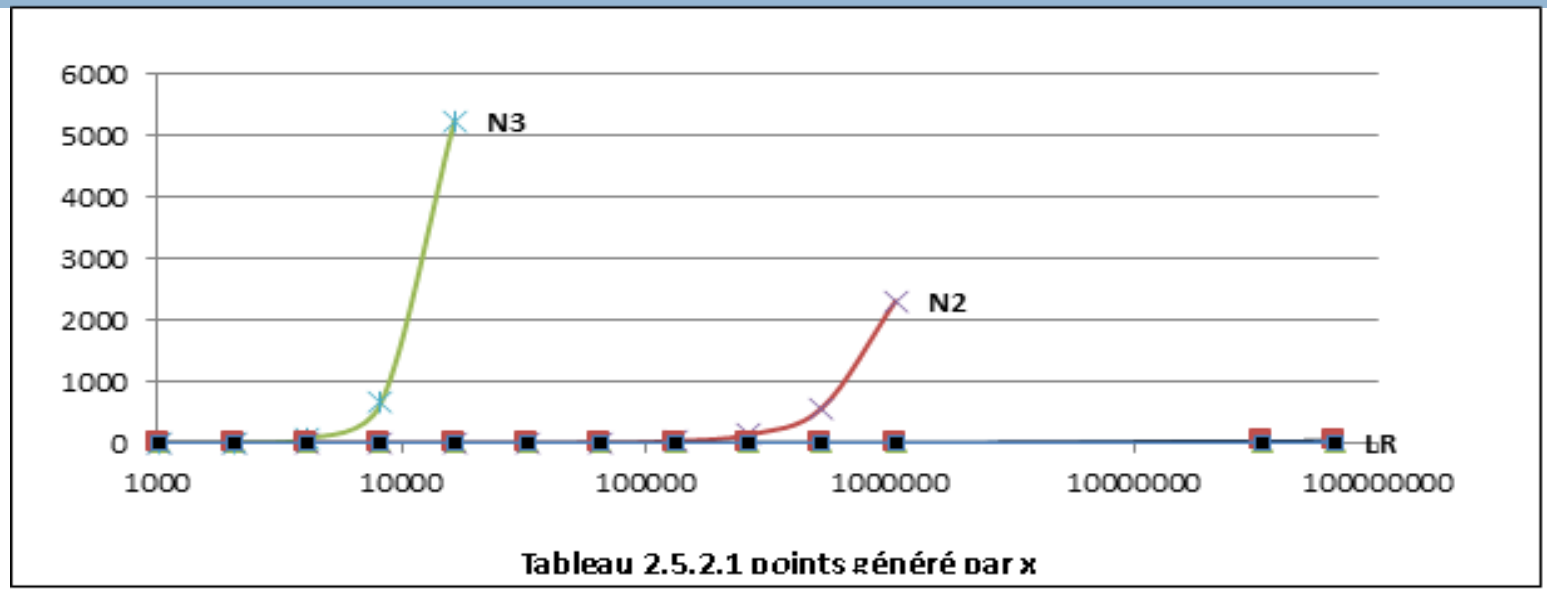
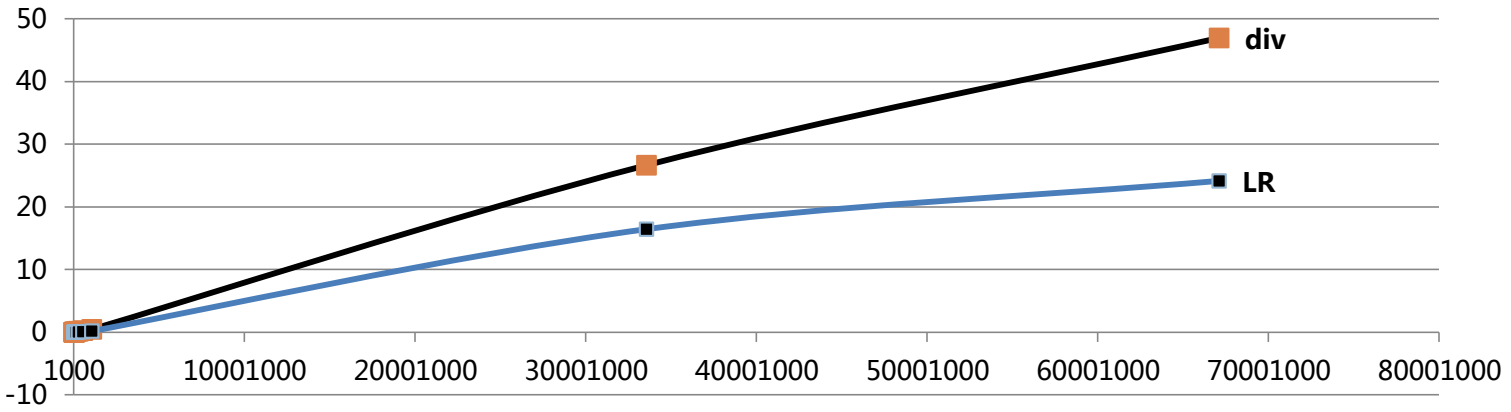
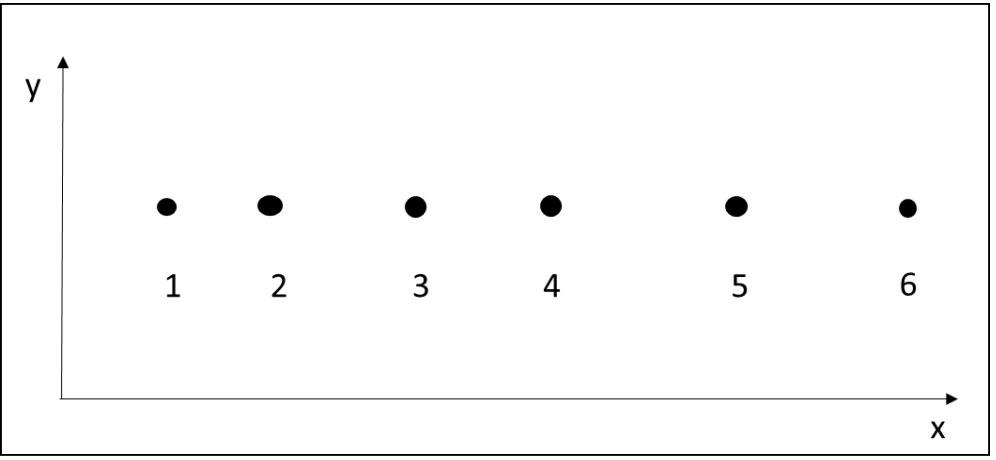


Tableau 2.5.2.1 points généré par x



Pire cas pour LR et diviser-pour-régner



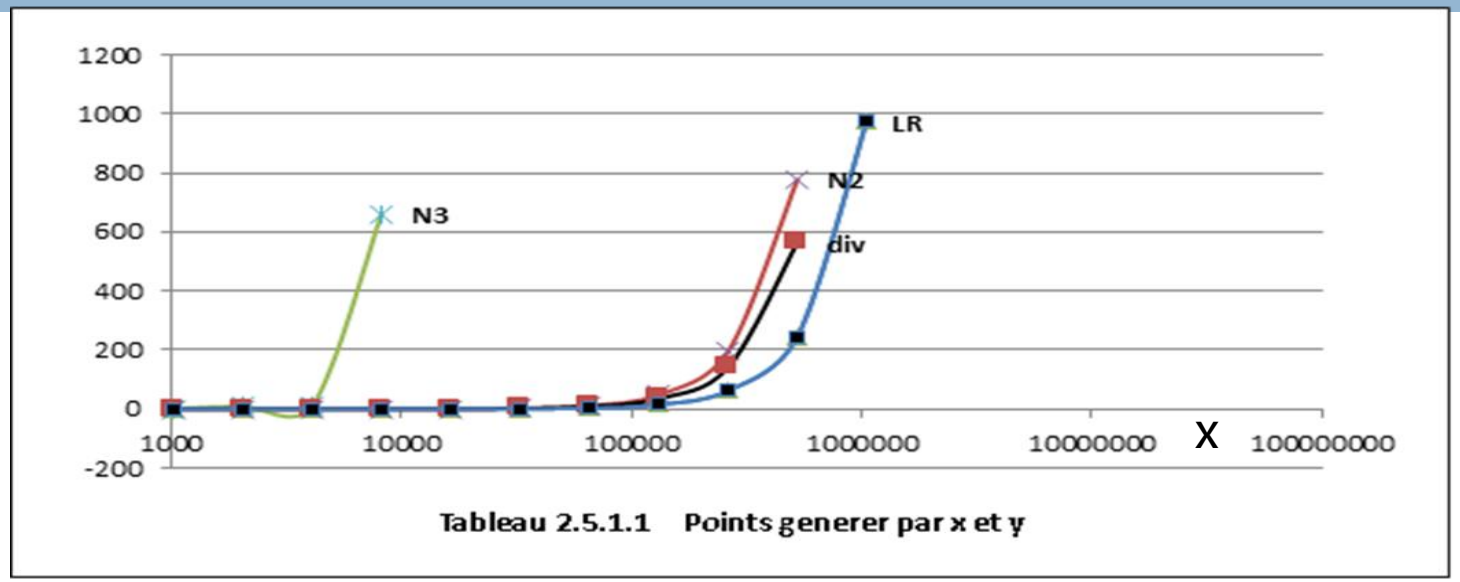
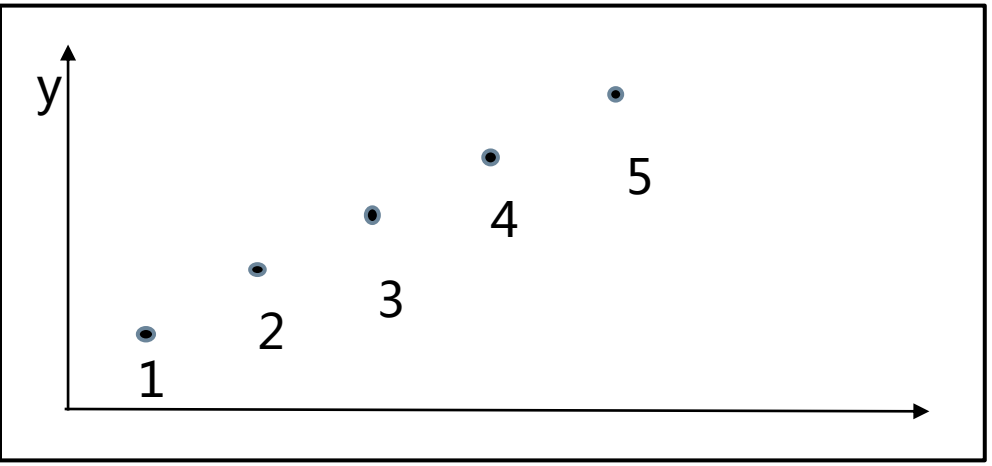
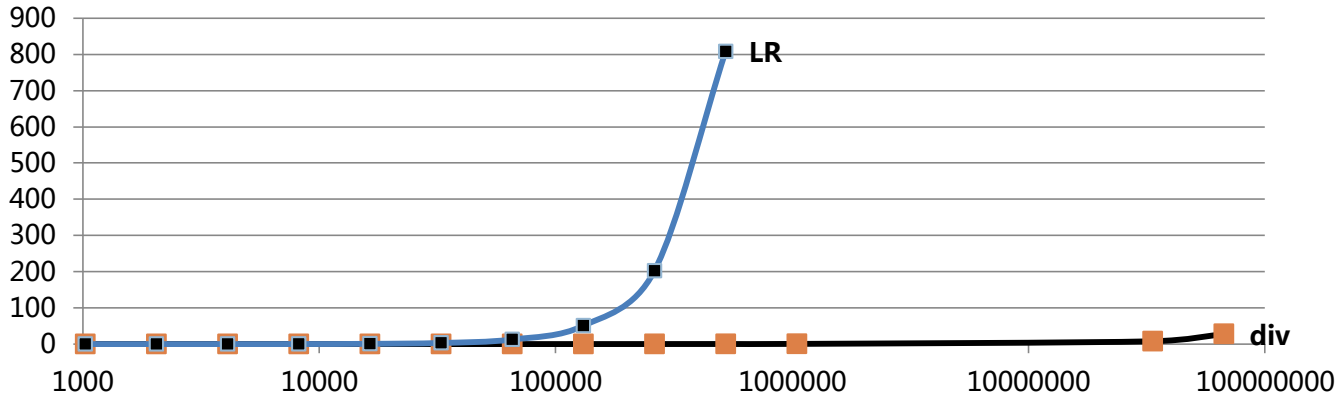


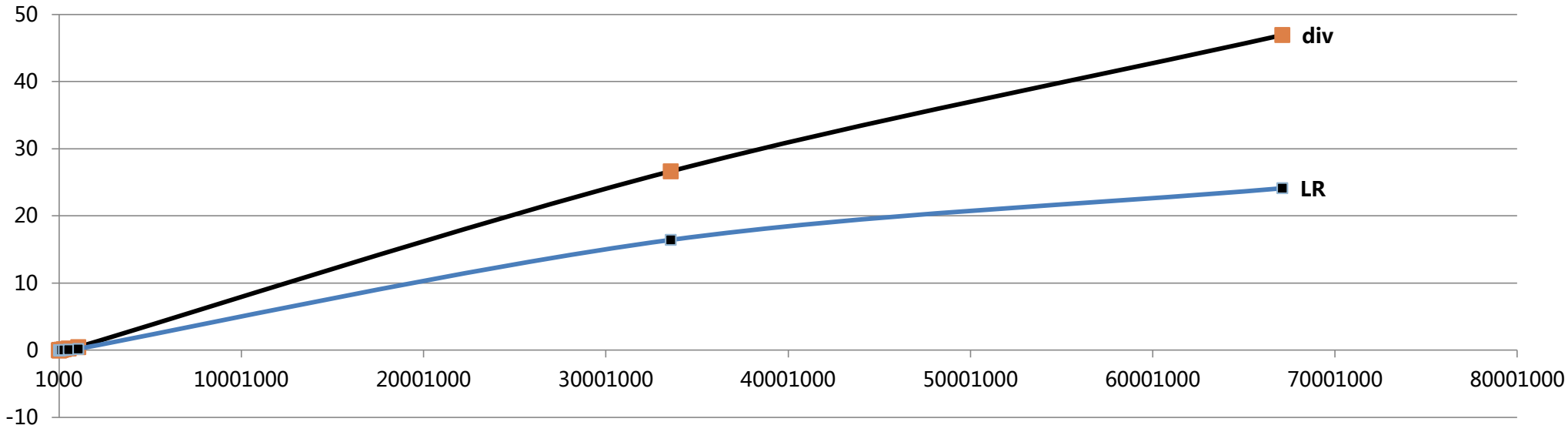
Tableau 2.5.1.1 Points generer par x et y





Conclusion

- Cas normal: nous pouvons conclure que l’ algorithme de Left-Right est la meilleur solution.



III. Parallel

- OpenMP
- MPI pour n^2
- OpenMP + MPI

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.

3.1.1 L'algorithme de complexité $O(n^3)$

- **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**
- calcule la surface entre i et j avec cette hauteur minimale
- On compare les surfaces et trouve la surface maximale
- **Renvoyer** max_surface

Optimisation et résultat

- 1. schedule(dynamic, chunk)
- 2.réduction (operation : variable)

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

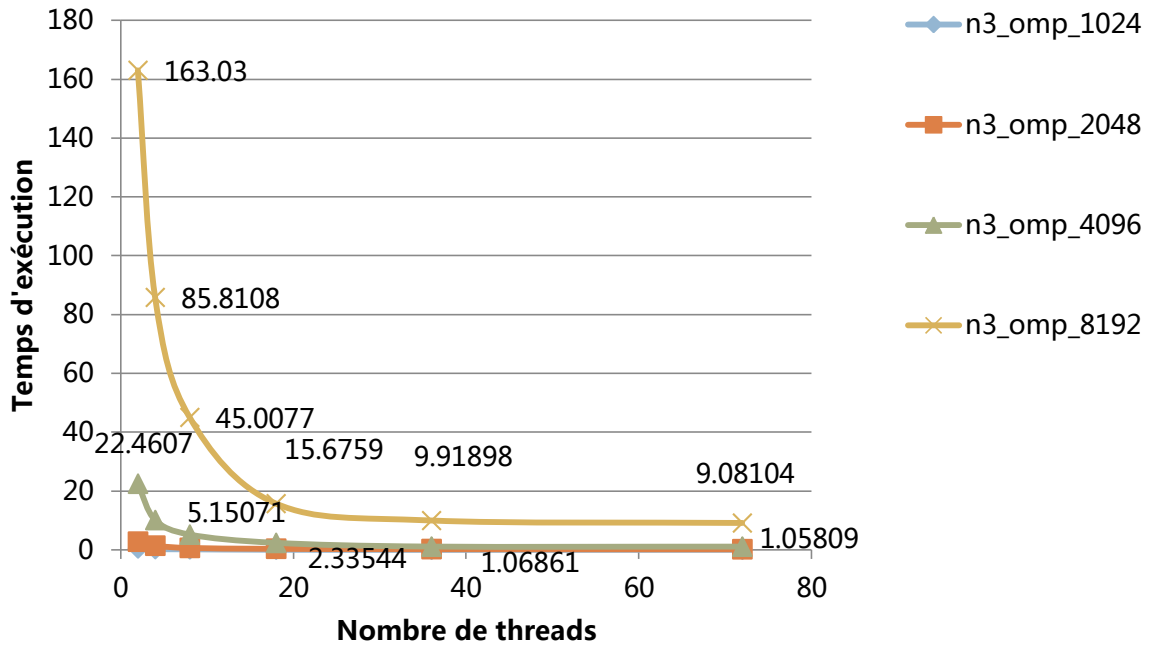


Tableau 3.1.1.1 - Temps d'exécution de l'algorithme n3

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

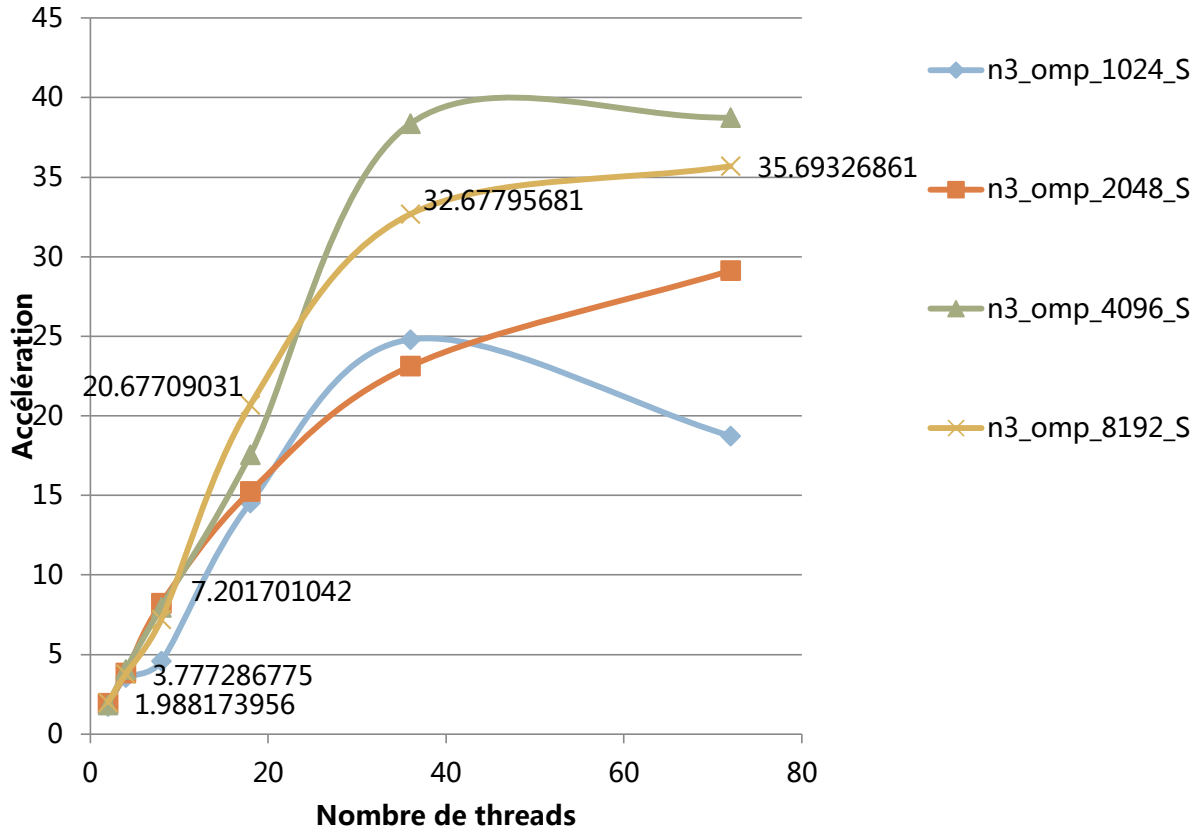


Tableau 3.1.1.2 - Accélération de l'algorithme n3 pour openMP

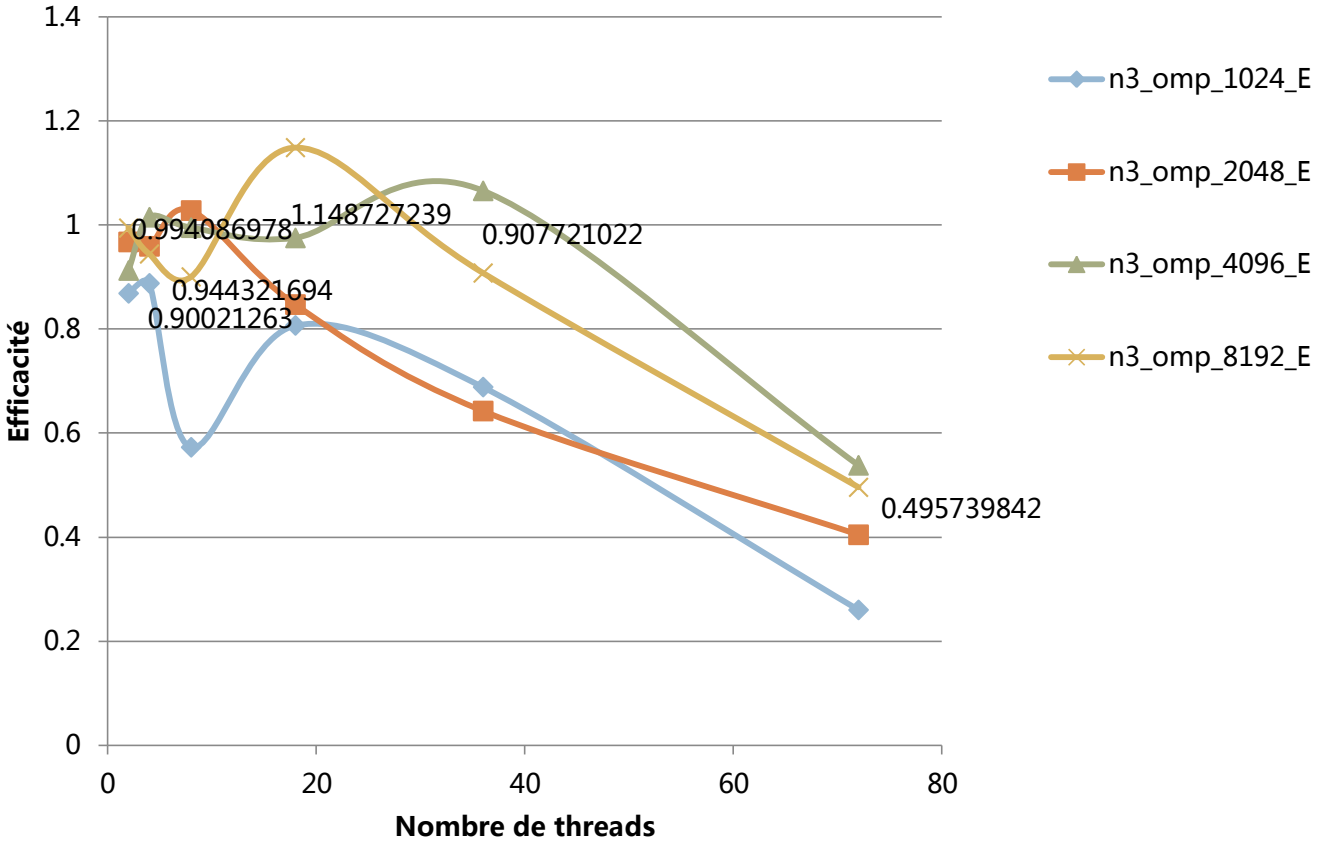


Tableau 3.1.1.3 - Efficacité de l'algorithme n3 pour openMP

3.1.2 L'algorithme de complexité O(n²)

- **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 i+1
- **Pour** tous les points j après le point i
- On mise 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 maximal
- **Renvoyer** max_surface

Optimisation et résultat

- la clause schedule(dynamic,10)
- reduction(max:max_surface)

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

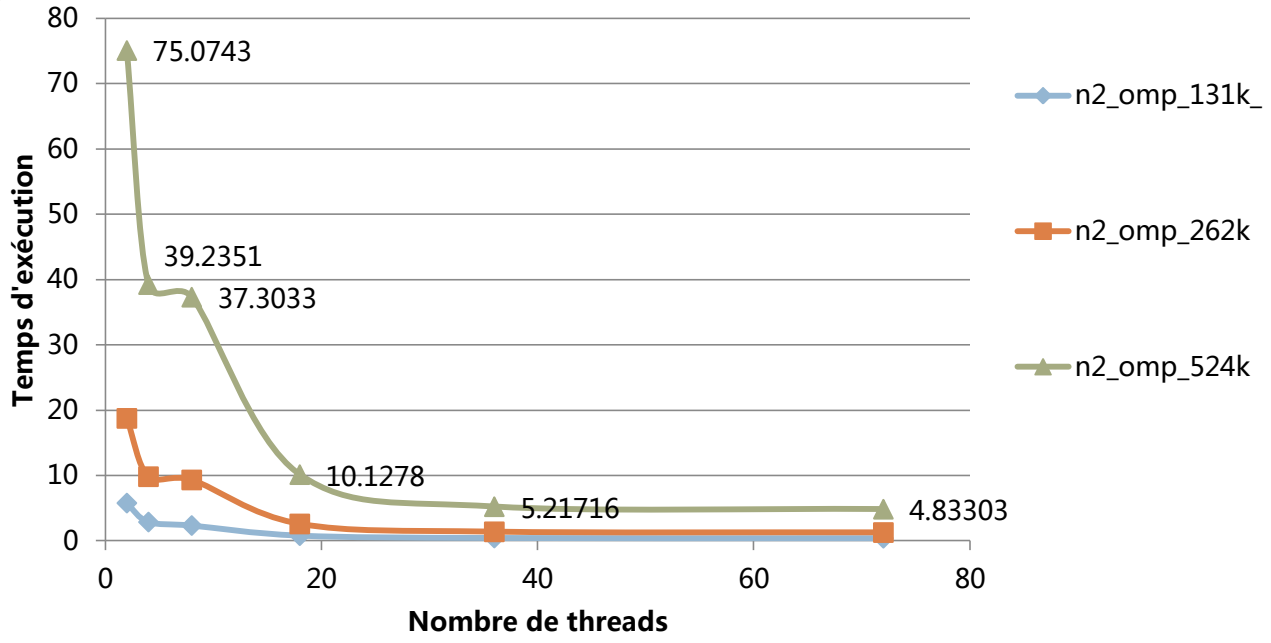


Tableau 3.1.2.1 - Temps d'exécution de l'algorithme n2

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

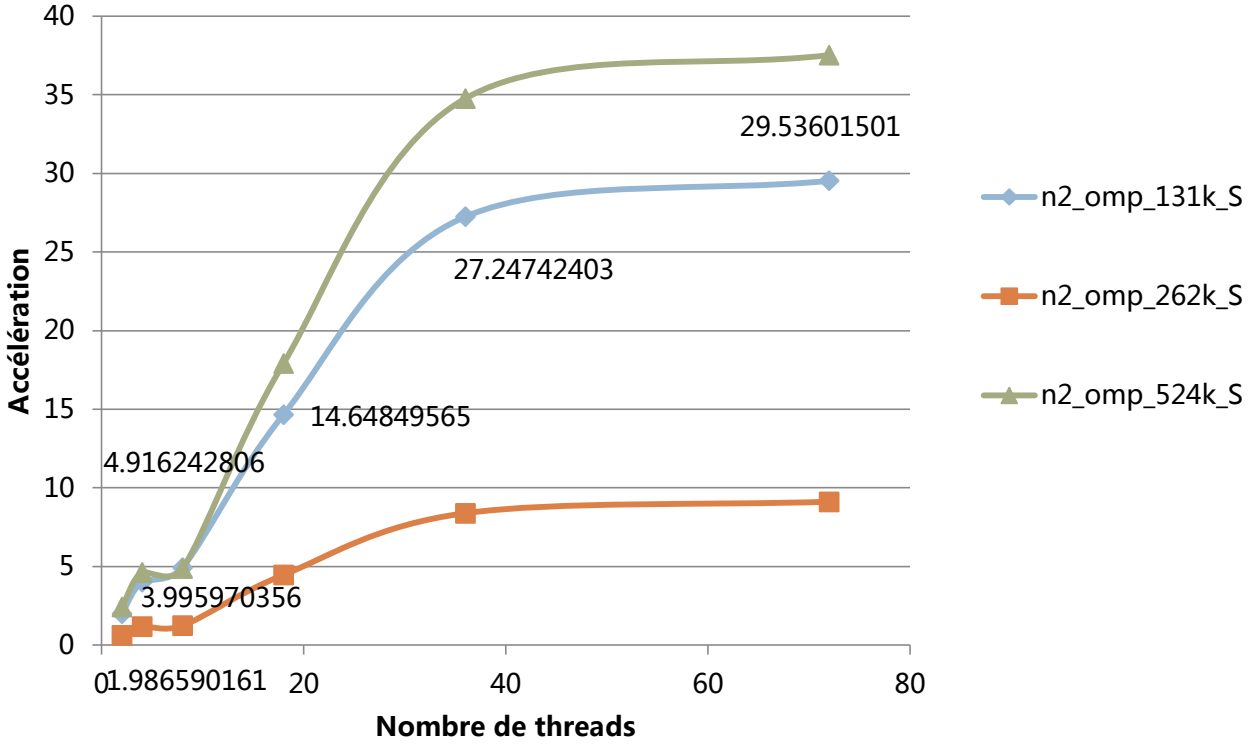


Tableau 3.1.2.2 - Accélération de l'algorithme n2 pour openMP

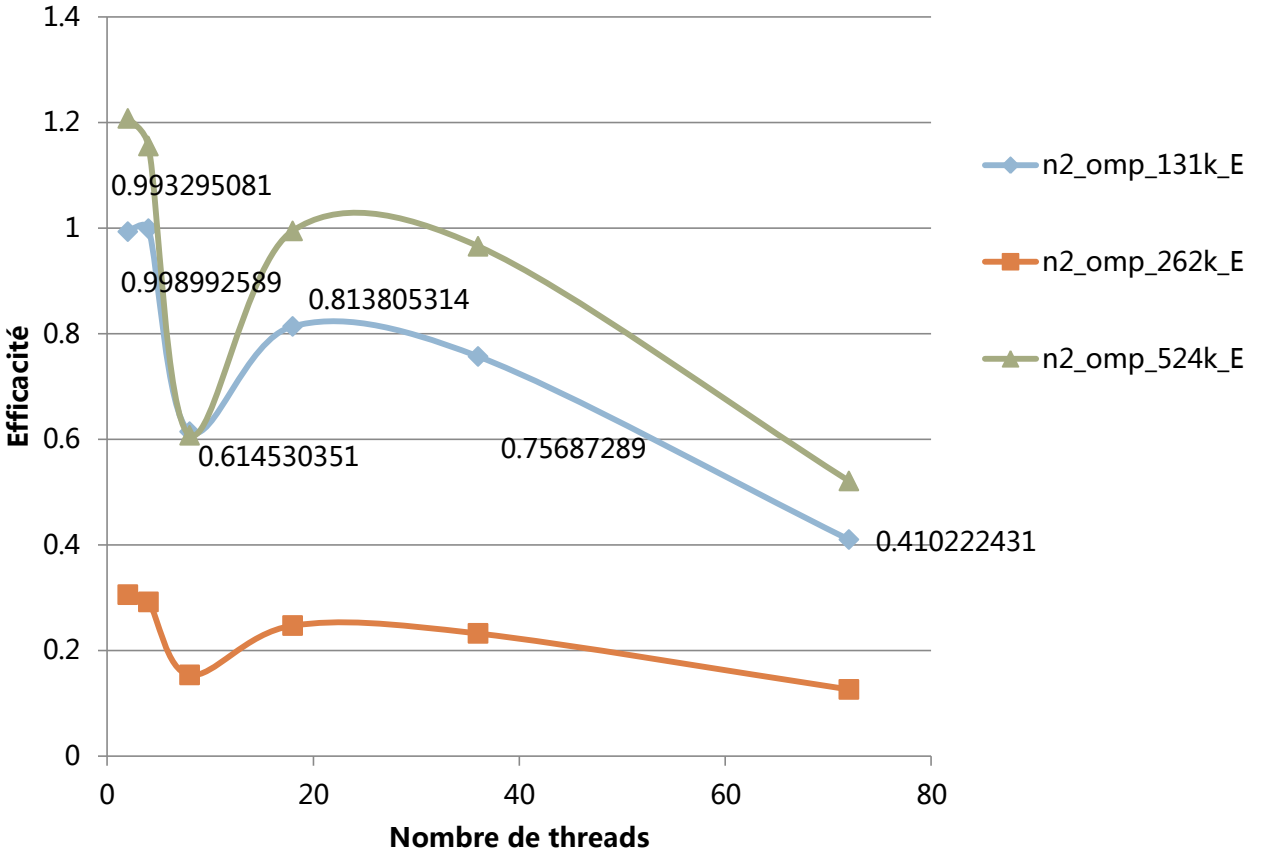


Tableau 3.1.2.3 - Efficacité de l'algorithme n2 pour openMP

3.1.3 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
- **#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 maximal
- **Renvoyer max_surface**

Optimisation et résultat

- la clause schedule(dynamic,10)
- la clause reduction(max:max_surface)

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

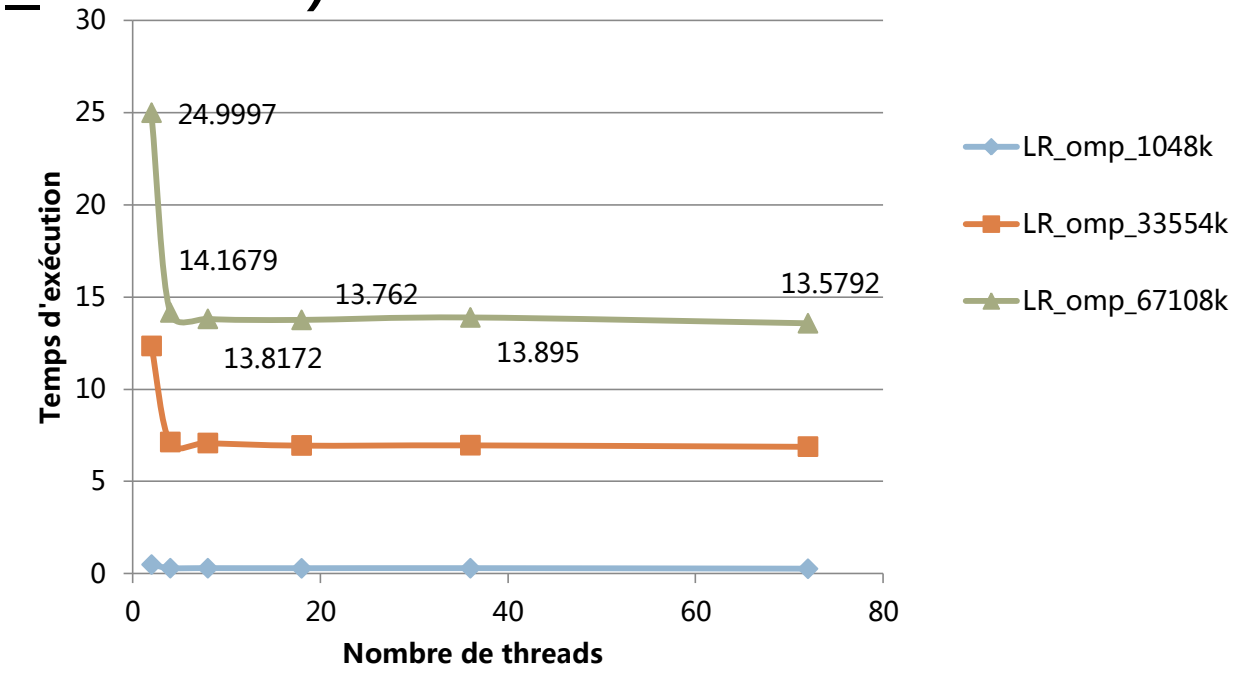


Tableau 3.1.3.1 - Temps d'exécution de l'algorithme LR pour openmp

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

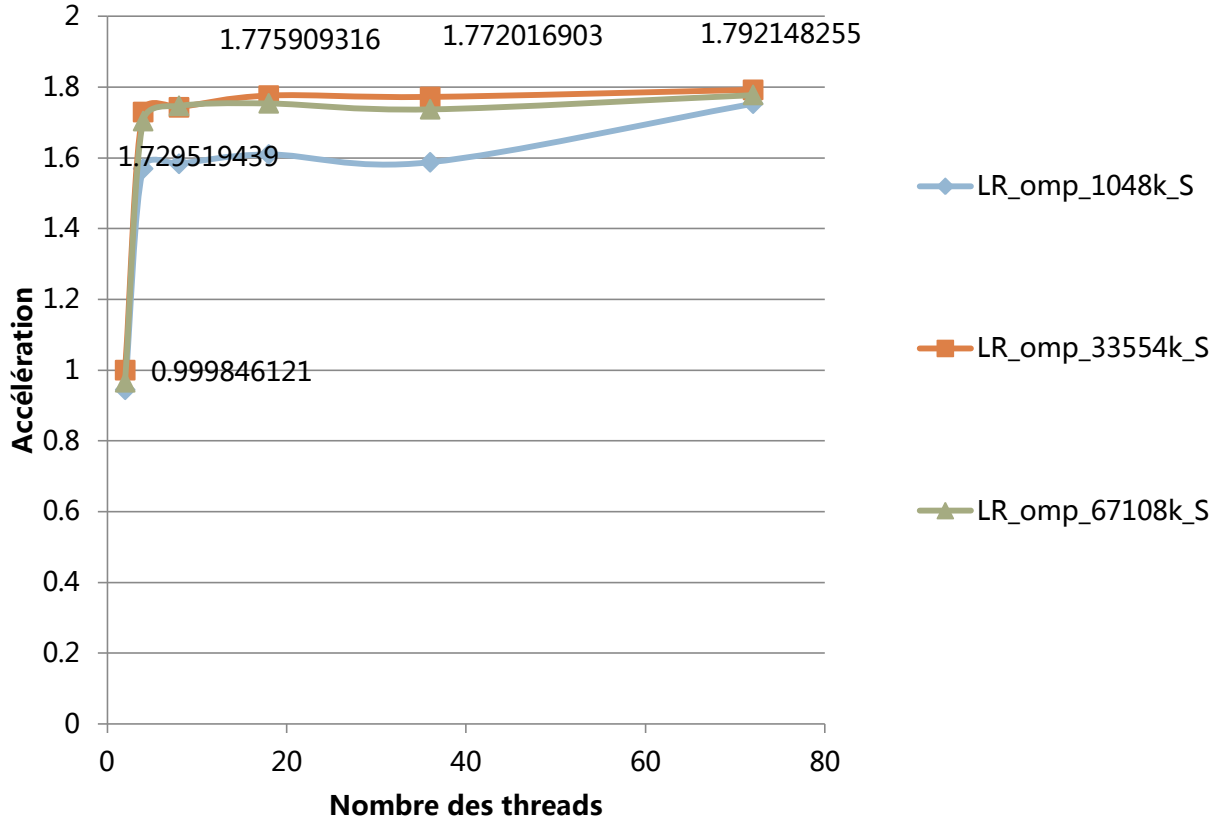


Tableau 3.1.3.2 - Accélération de l'algorithme LR pour openMP

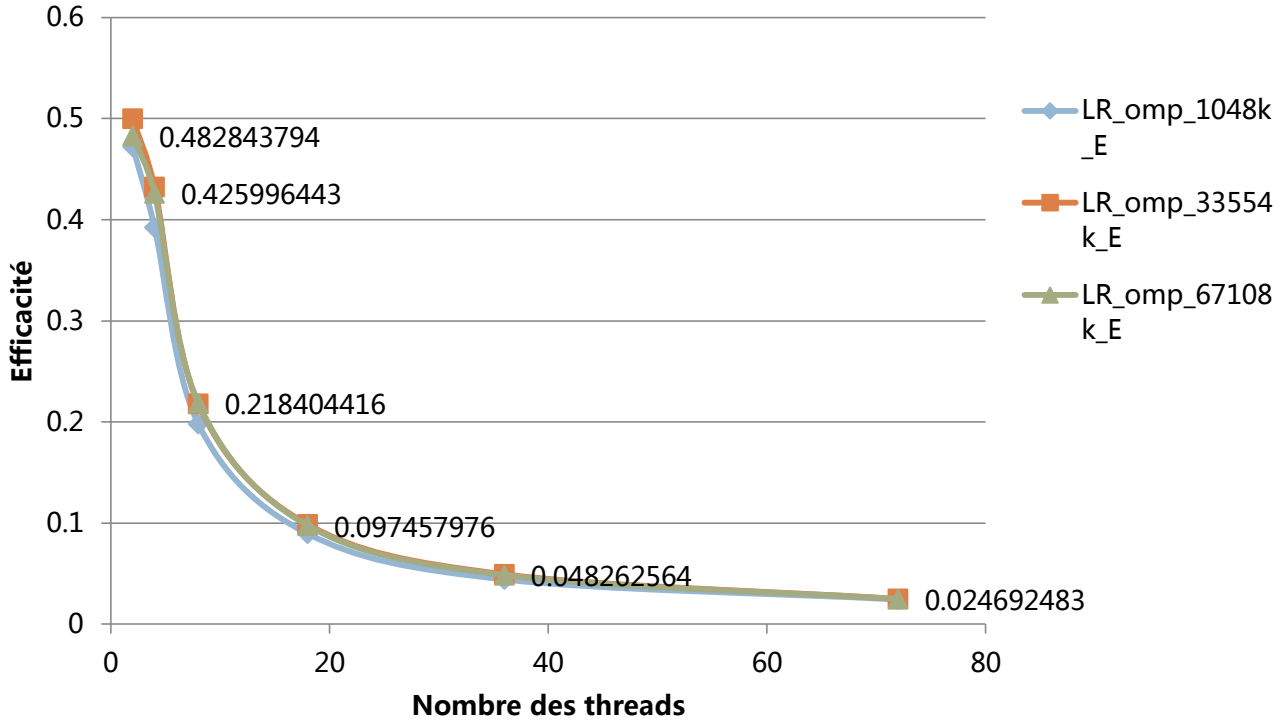


Tableau 3.1.3.3 - Efficacité de l'algorithme LR pour openMP

3.1.4 L'algorithme de diviser-pour-régner

- **Entrée:** table des points table_points, nombre des points n
- **Sortie:** la surface maximale
- Récursif (left,right) :
- **#pragma omp parallel**
- **#pragma omp single**
- **Initialiser** max_surface ← 0, surface_left ← 0, surface_right ← 0, left ← 0 et right ← l
- length ← right - left
- **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**

Optimisation et résultat

- 1. Seuil
- 2. Untied

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

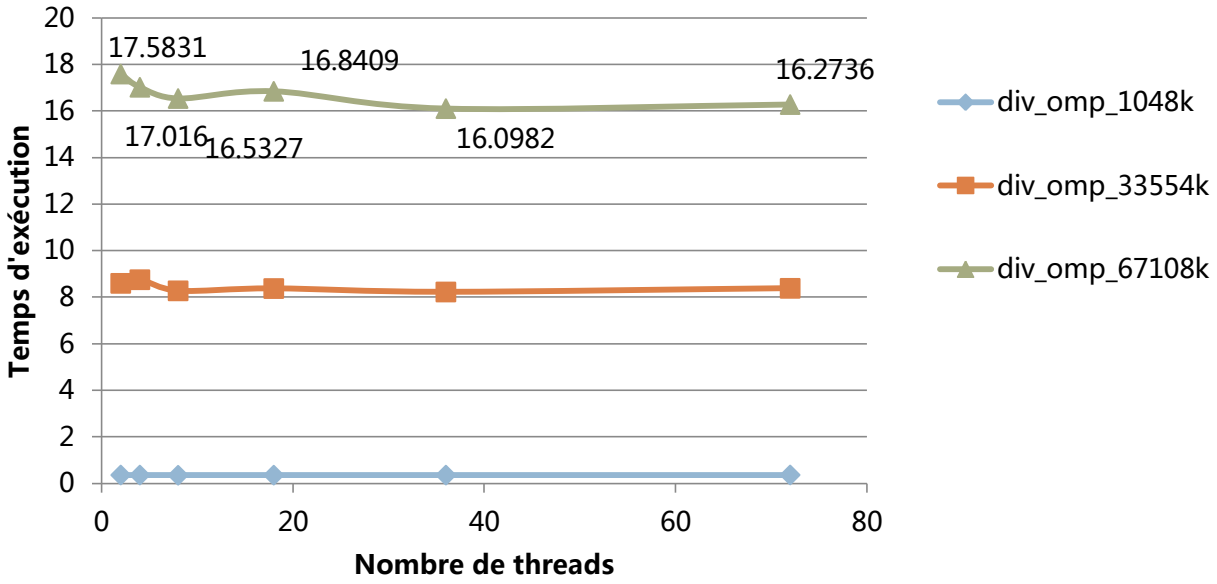


Tableau 3.1.4.1 - Temps d'exécution de l'algorithme Diviser-pour-régner pour openMP

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

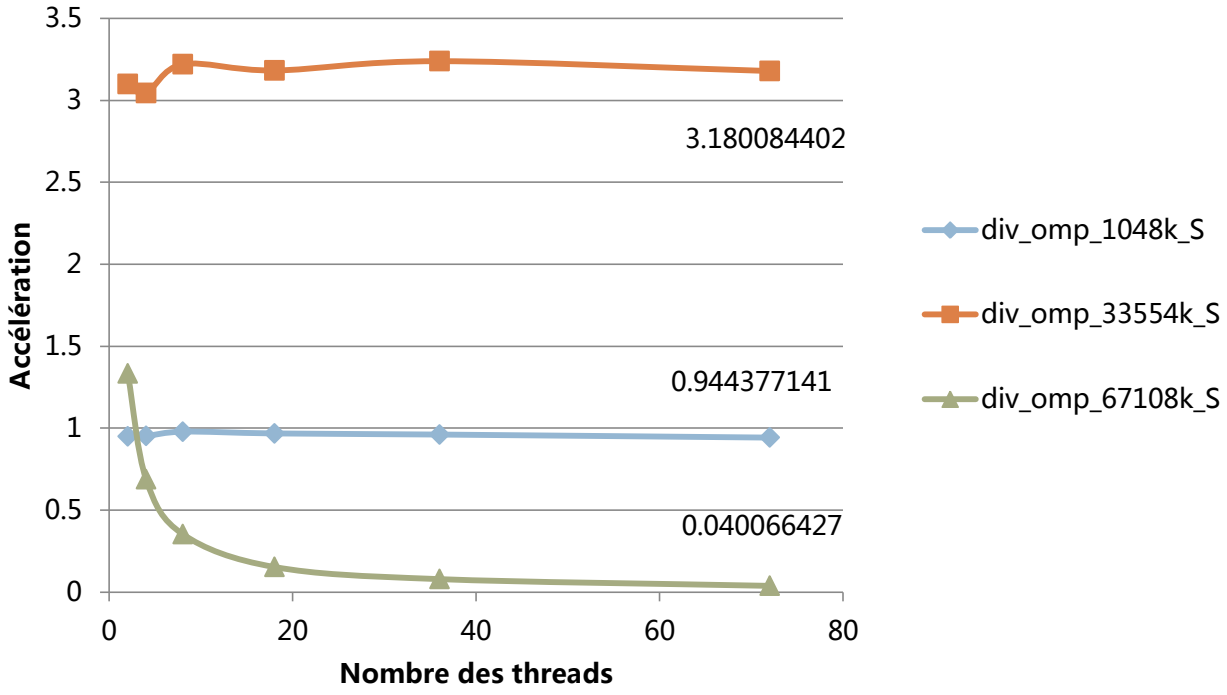


Tableau 3.1.4.2 -Accélération de l'algorithme Diviser-pour-régner pour openMP

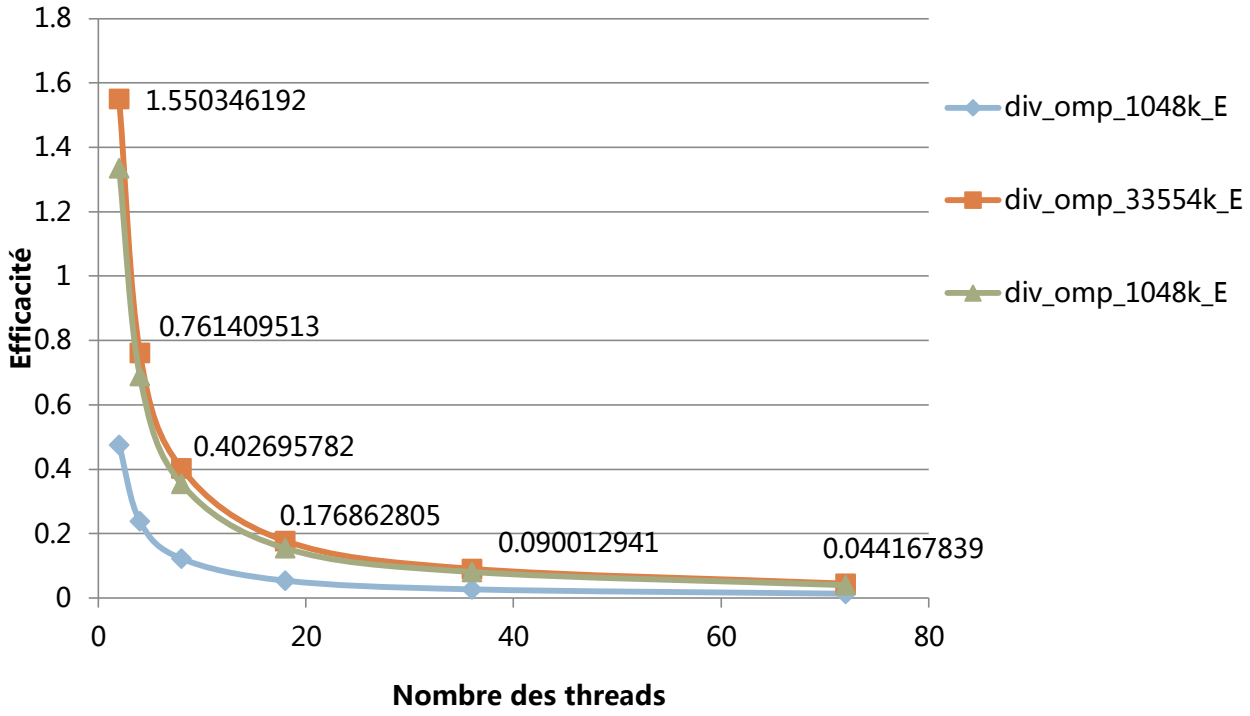


Tableau 3.1.4.3 -Efficacité de l'algorithme Diviser-pour-régner pour openMP

Conclusion

- Avantage
- Inconvénient

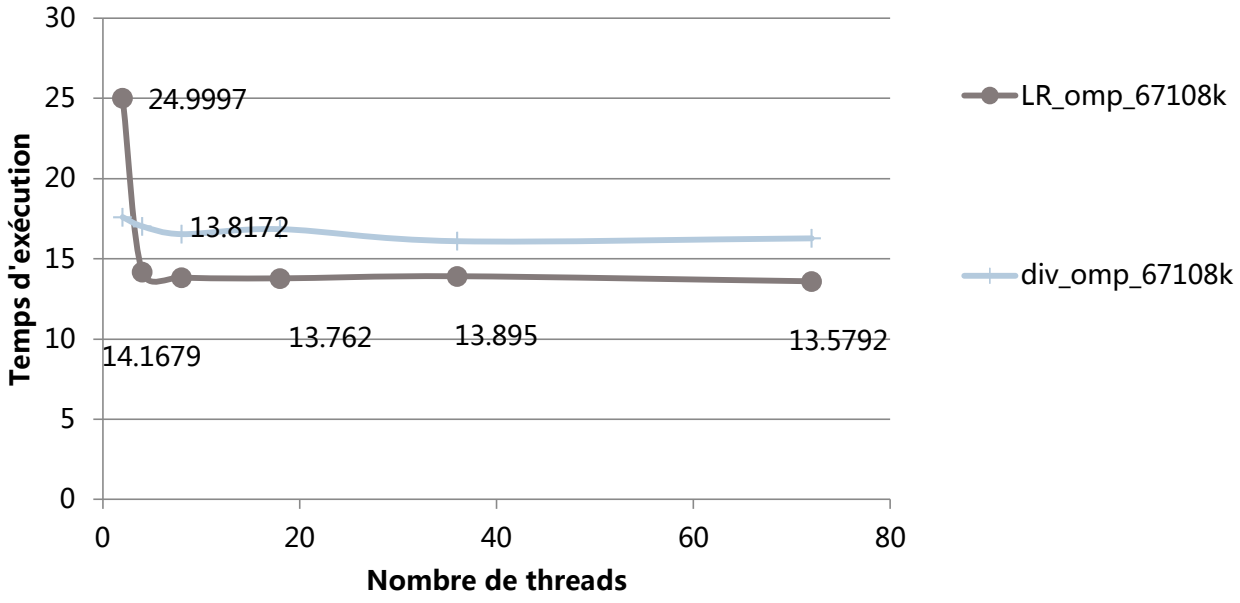


Tableau 3.1.5.1 - Temps d'exécution des quatre algorithmes pour openMP

- Résultat

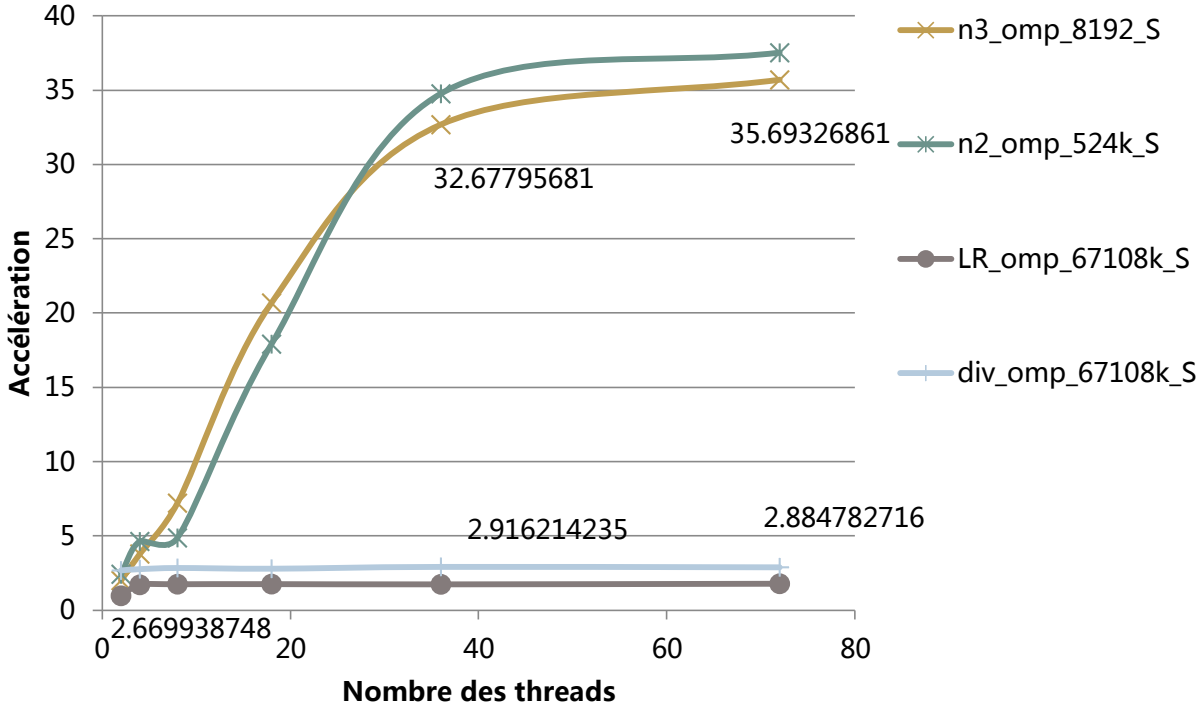


Tableau 3.1.5.2 - Accélération des quatre algorithmes pour openMP

3.2 MPI

- Nous utilisons donc la bibliothèque MPI qui permet de réaliser facilement un programme distribué qui peut utiliser plusieurs machines pour les calculs.

- MPI permet de se passer de la gestion de la couche réseaux du programme gère automatiquement la création des processus répartis sur les machines

3.2.1 Parallélisme à équilibrage de 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 $O(n^2)$
- **Finpour**
- **Sinon**
  - **pour** i de tailleplage x rang à n
  - L'algorithme de complexité $O(n^2)$
  - **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 Des méthodes du parallélisme MPI

- Parallélisme à équilibrage de charge dynamique
  - Parallélisme à maître–esclave

Résultat

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

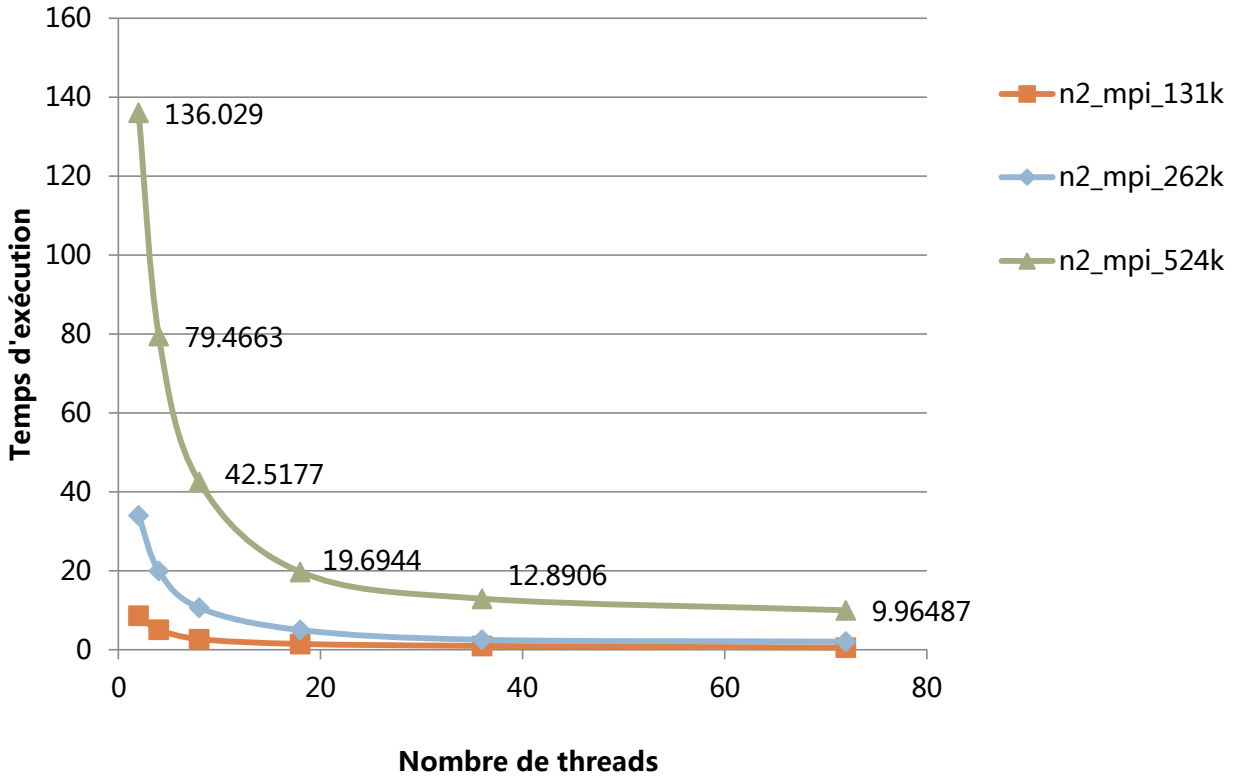


Tableau 3.2.5.1 - Temps d'exécution de l'algorithme n2 pour MPI

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

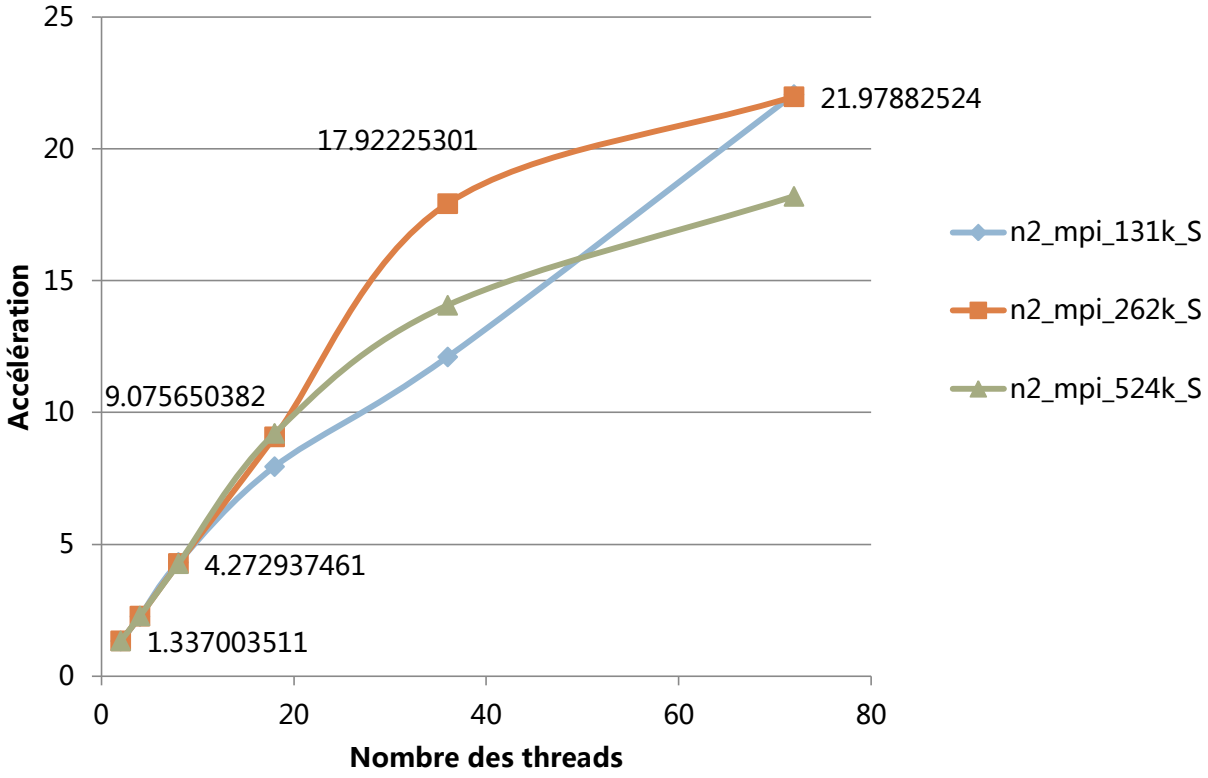


Tableau 3.2.5.2 -Accélération de l'algorithme n2 pour MPI

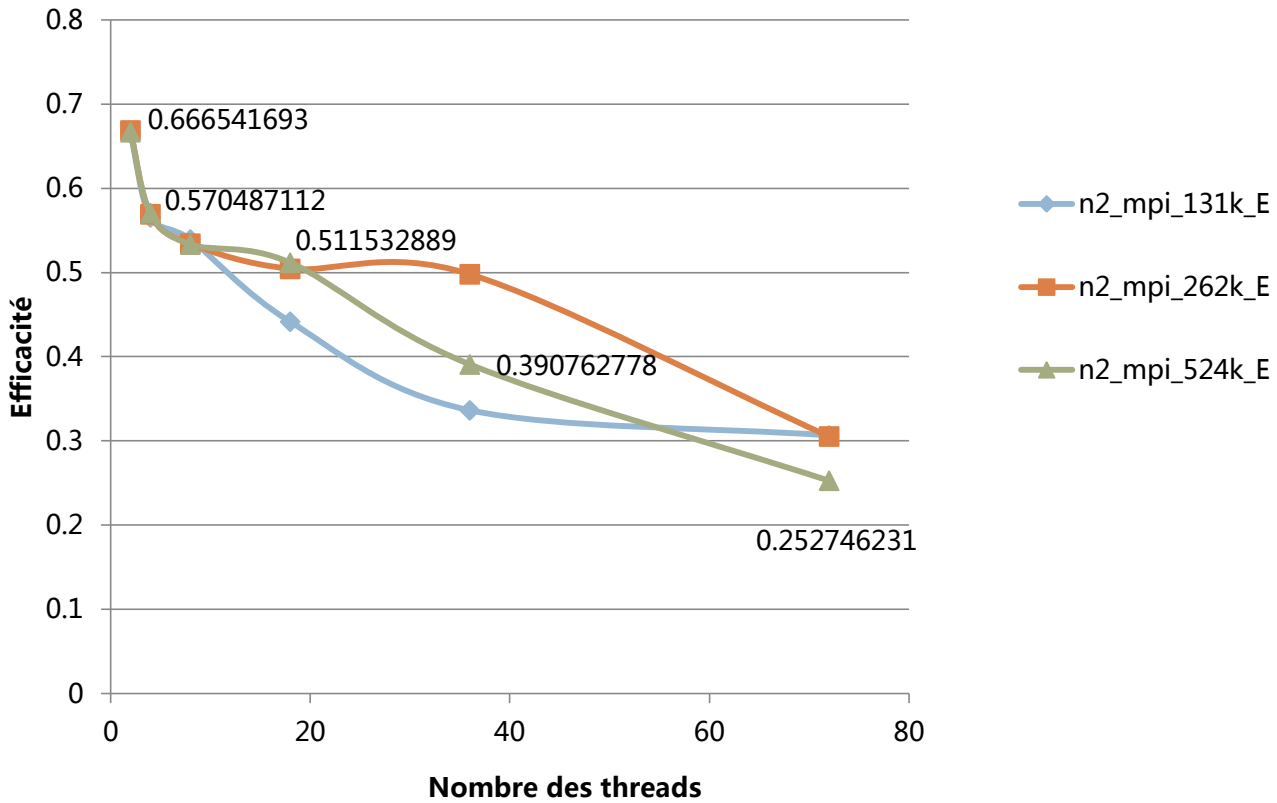


Tableau 3.2.5.3 -Efficacité de l'algorithme n2 pour MPI

Conclusion

- MPI avantage et inconvénient

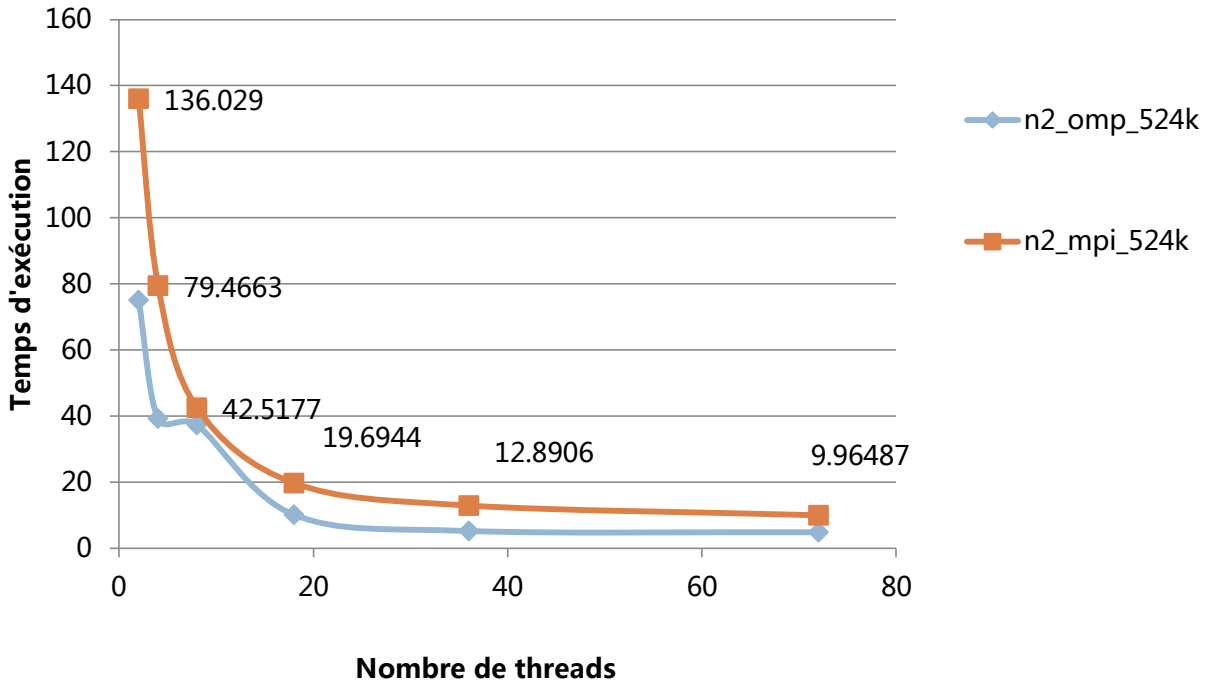


Tableau 4.3.1 - Temps d'exécution de l'algorithme n2 en omp et mpi

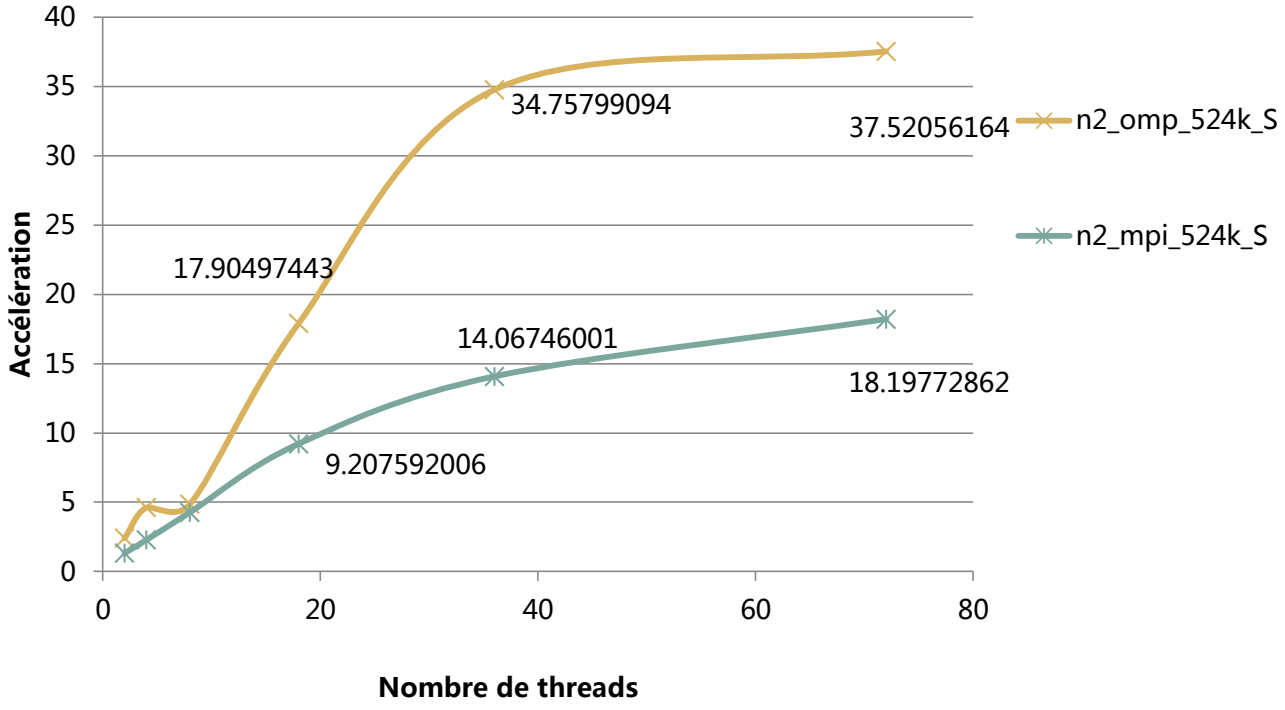


Tableau 4.3.2 -Accélération de l'algorithme n2 en omp et mpi

3.3 OpenMP + MPI

- Multi-core
- Multi-threads
- MPI augmente la performance avec les processeurs
- OpenMP augmente la performance avec les threads
- Réalisation:

  Ajouter les codes #pragma omp for avant la boucle première for dans la base de l’ algorithme de MPI.

IV. Conclusion

- 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.

- 3. Les algorithmes séquentiels:

  l'algorithme LR est la meilleure solution.

- 4. Les algorithmes parallèles:

  l'algorithme LR a le meilleur temps d'exécution, mais le meilleur effet du parallélisme est l'algorithme n3 et n2.

V. Bilan

- 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.

Merci beaucoup!