# Rapport Programmation Parallèle L3S6 Informatique

DAVID Aurélien

HOCHBERGER Dylan

## FONCTION MAIN()

Afin d'être en mesure d'utiliser MPI, nous avons procédé à quelques changements.

### MAIN()

Nous avons simplement rajouté `main(int argc, char **argv)` , un `MPI_INIT()` en début de fonction ainsi qu'un `MPI_Finalize()` en fin de fonction par convention. Nous utilisons également `MPI_Scatter()` et `MPI_Gather()` dans le `main()` afin de découper le tableau et récupérer les blocs des autres processus.

### MPI_Scatter()

Nous avons longuement réfléchi à la meilleure manière d'utiliser `Scatter()` .
Etant donné qu'à la fin du `MPI_Scatter()` , chaque bloc était séparé selon le nombre de processus, il fallait pouvoir envoyer et récupérer certaines lignes des autres blocs.
Ainsi, plutôt que séparer le tableau t1 du premier processus et placer les blocs des autres

processus directement au début du tableau, nous les avons placé à l'endroit du découpage dans les autres processus.
Nous utilisons donc la taille complète, divisée par le nombre de processus `HM/size` , et plaçons les blocs en fonction du processus : `HM/size*rank` . Nous avons donc une fonction `MPI_Scatter()` de la forme `MPI_Scatter(t1, HM*LM/size, MPI_CHAR, t1[HM/size*rank], HM*LM/size, MPI_CHAR, 0, MPI_COMM_WORLD);` . Etant donné que chaque processus fait les calculs uniquement sur son bloc, il n'était pas nécessaire de découper le tableau à chaque itération, nous avons donc un seul `MPI_Scatter()` suivant l' `init(t1);` de tout le programme. C'est le processus 0, processus qui initialise le tableau qui s'occupera d'envoyer de découper et d'envoyer le tableau t1.

MPI_Gather()

`MPI_Gather()` est l'exact inverse de notre `MPI_Scatter()` . Nous regroupons les blocs du tableau en fonction de leur position dans les différents processus.
Nous avons donc une fonction de la forme `MPI_Gather(t1[HM/size*rank], HM*LM/size, MPI_CHAR, t1, HM*LM/size, MPI_CHAR, 0, MPI_COMM_WORLD);` .
Nous récupérons les blocs des différents processus et les envoyons dans le tableau t1 du processus 0. Etant donné qu'à chaque sauvegarde nous avons besoin du tableau entier, il y a donc un `MPI_Gather()` par sauvegarde. Sans ce `MPI_Gather()` , nous n'aurions pas la dernière mise à jour du tableau et l'affichage serait faux.

## FONCTION CALCNOUV()

Nous avions d'abord réfléchi à envoyer les lignes manquantes dans la fonction `main()` , autour des appels à la fonction `calcnouv()` , mais nous avons très vite remarqué que par soucis d'optimisation de code et de performances, il serait plus simple de le faire directement dans `calcnouv()` .
Nous avons également rajouté une condition afin de vérifier le nombre de processus et ne pas envoyer de ligne lorsque le programme ne lance qu'un seul processus.

MPI_Issend()

Afin d'optimiser le programme et de faire les calculs pendant l'envoi des données, nous avons décidé d'utiliser un envoi non-bloquant. Nous avons ainsi deux conditions `if` permettant aux processus d'envoyer leur première ou dernière ligne selon les besoins. Etant donné que le processus 0 n'a les lignes que de 0 à HM/size-1, il lui est nécessaire de recevoir la ligne HM/size du processus suivant afin de calculer sa dernière ligne. De même, le processus 1 aura donc un bloc allant de `HM/size*rank` à `HM/size*`

`(rank+1)-1` , étant donc les bornes de chaque bloc car pour HM=1200 lignes, le tableau ira donc de t[0] à t[1199].
Nous avons donc 2 conditions `if` afin de savoir dans quelle boucle le processus devra passer. Si les processus ne sont pas le dernier, ils enverront la dernière ligne de leur partie du tableau, `t[(HM/size)*(rank+1)-1]` au processus suivant. L'appel à Issend ressemble donc à cela : `MPI_Issend(t[(HM/size)*(rank+1)-1], LM, MPI_CHAR, rank+1, 0, MPI_COMM_WORLD, &requestSend);` .

Les processus n'étant pas le premier passeront dans le 2e `if` afin d'envoyer leur première ligne au processus précédent. Ainsi les processus n'étant ni le premier, ni le dernier passeront donc dans les deux conditions afin d'envoyer et leur première, et leur dernière lignes.

MPI_Irecv()

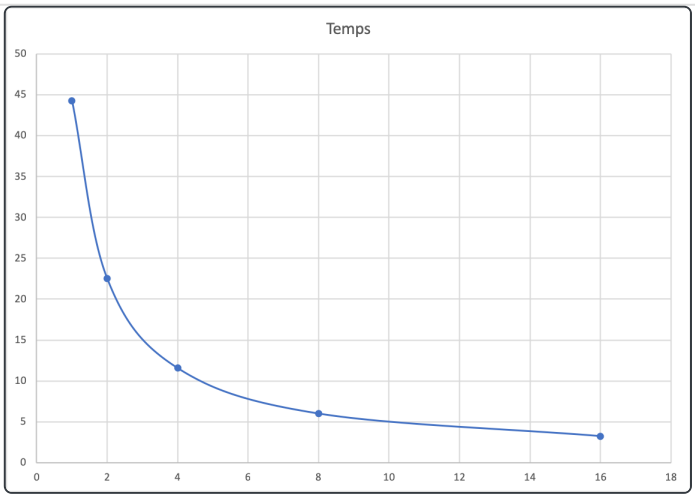
De même que pour la fonction `MPI_Issend()` , nous utilisons une réception non-bloquante pour que les processus reçoivent les lignes nécessaires au bon déroulement du programme pendant qu'ils commencent les calculs de leur bloc.
Ainsi, les processus n'étant pas le premier recevront la dernière ligne du processus précédent, et les processus n'étant pas le dernier recevront la première ligne du processus suivant.

MPI_Wait()

Pendant la boucle de calcul, les processus n'étant pas le premier sauteront la première ligne qu'ils calculeront à la sortie de la boucle `for` . Ainsi le processus peut déjà calculer son bloc sans avoir à attendre que la dernière ligne du bloc précédent soit présente pour commencer sa boucle.
De même, les processus n'étant pas le dernier calculeront leur bloc, et vérifieront à la fin du bloc qu'ils ont bien reçu la première ligne du processus suivante afin de calculer leur dernière ligne.

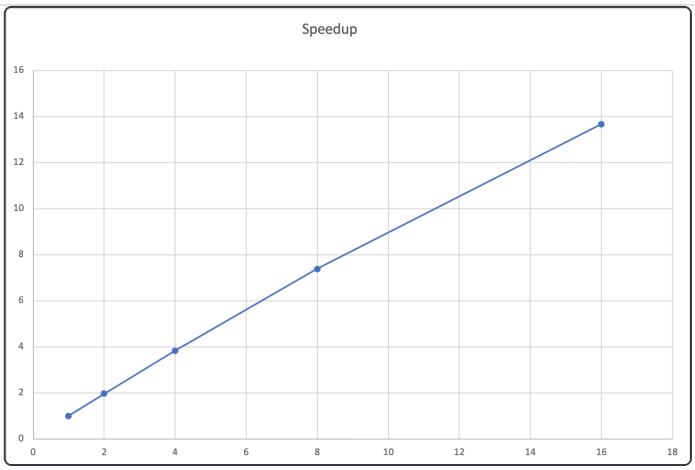
## PERFORMANCES

### Temps



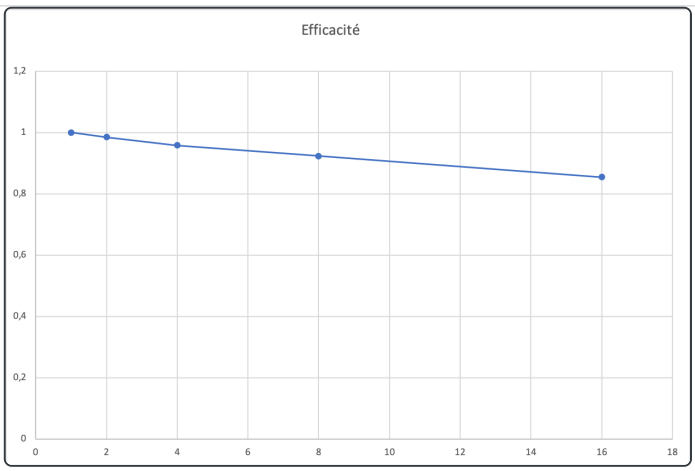

Pour nos mesures de performances, nous avons effectué 10 mesures pour chaque cas (1, 2, 4, 8 et 16 coeurs) puis fait la moyenne de ces valeurs pour avoir des résultats les plus cohérents possible. On constate sur notre courbe du temps que plus nous avons de coeurs s'attelant sur le programme, moins ce dernier prenait de temps à se terminer, avec un programme 16 fois plus rapide pour 16 coeurs qu'avec un.

### Speedup




Grâce à ces mesures de temps, nous avons donc calculé le speed-up de notre programme (avec la formule Sp=T1/Tp) et nous pouvons en conclure que l'accélération est quasiment linéaire.

## Efficacité




Nous avons également calculer l'efficacité de notre programme (grâce à la formule Sp/p) et on peut constater que son efficacité s'approche de 1, même si elle est logiquement en baisse lorsque le nombre de coeurs qui exploitent le programme augmente.

## Données utiles

| Nb de proc | Temps | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Speedup | Efficacité |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 48,7238914 | 49,2256402 | 48,9196796 | 47,8872757 | 47,8265469 | 48,9861141 | 49,4136775 | 49,0843562 | 49,480398 | 48,7309713 | 47,6842542 | 1 | 1 |
| 2 | 24,7488014 | 24,96945 | 24,8931408 | 24,4367464 | 24,6190593 | 24,4866501 | 24,6954994 | 24,7301857 | 24,8500109 | 24,9053002 | 24,9019716 | 1,96873741 | 0,98436871 |
| 4 | 12,7103121 | 12,6813687 | 12,6389219 | 12,6663427 | 12,6447112 | 12,5983891 | 12,7714444 | 13,0371956 | 12,5944159 | 12,5540767 | 12,916255 | 3,83341423 | 0,95835356 |
| 8 | 6,59787172 | 6,6250855 | 6,646937 | 6,6673827 | 6,5214919 | 6,4061789 | 6,6774466 | 6,4066002 | 6,6403931 | 6,7128985 | 6,6743028 | 7,38478913 | 0,92309864 |
| 16 | 3,56451359 | 3,5239358 | 3,6043898 | 3,5719497 | 3,5673869 | 3,6004716 | 3,5965633 | 3,5791404 | 3,5512609 | 3,5289474 | 3,5210901 | 13,6691557 | 0,85432223 |

Pour compiler le programme: mpicc -O3 -march=native jeudelavie.c

Pour exécuter:
mpirun -n 16 -tune tune.txt -hostfile hostfile.txt a.out

Pour vérifier que le programme donne les bons résultats, nous comparons le fichier du programme séquentiel fourni pour le projet avec notre fichier de sortie: diff jdlv.out test.out. (où test.out est notre résultat)

Exemple d'un hostfile pour 16 coeurs:
localhost slots=4
uds-501611 slots=4
uds-501612 slots=4
uds-501614 slots=4

## CONCLUSION

Ce petit projet était très intéressant pour nous aider à comprendre comment fonctionnait les différents types d'envoi et de réception avec MPI. Explorer différentes solutions afin de trouver la meilleure solution mêlant optimisation de code et de performances est toujours une facette passionante de la programmation.