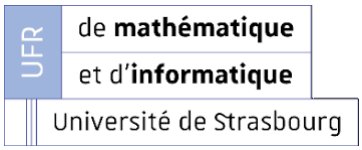
Rendu de projet

« Algorithme de Darboux et
Modèles Numériques de Terrain »

Programmation parallèle

UFR Math-Info - Université de Strasbourg

3ième année de Licence Informatique



Date de rendu : 22 avril 2022

Par

Matthieu FREITAG
et Paul MEYER

https://github.com/SchawnnDev/MNT

Index

1 Introduction

Dans ce projet, nous proposerons l'étude de l'algorithme de Darboux, appliqué aux modèles numériques de terrain (MNT).

Approche proposée

Le programme initial étant fourni, il nous est simplement demandé de paralléliser l'algorithme de Darboux (réalisé en C) via les outils suivants :

1. OpenMP.
2. MPI.

Ainsi, après chaque utilisation du programme, le temps de référence (programme non parallélisé) sera comparé au temps du programme parallélisé (modifiable selon les envies de l'utilisateur, cf. le `makefile`).
Naturellement, pour vérifier le bon fonctionnement de notre implémentation, une comparaison des résultats sera effectuée entre les programmes parallélisé/non parallélisé.

Dans un premier temps, nous indiquerons nos choix d'implémentation, puis, dans un second temps, nous étudierons les performances obtenues par notre algorithme parallélisé.

Note : Nous nous invitons à bien vous renseigner sur les fonctionnalités qu'offre le `makefile`. En effet, ce dernier est très permissif et rend le projet réellement personnalisable.

Implémentation 2

2.1 OpenMP

Lors de la parallélisation OpenMP de chaque boucle `for`, nous avons utilisé la clause `default(none)`. Elle oblige à spécifier explicitement les attributs de partage de données pour les variables référencées, ce qui rend les erreurs plus faciles à repérer. C'est `clang` qui nous l'a fait la remarquer via un `warning`.

En dehors de cette précision, l'implémentation reste plus ou moins trivial et nous ne voyons pas vraiment l'utilité de rentrer dans les détails. En revanche, nous allons tout de même brièvement répertorier notre implémentation :

1. Ligne `24` dans `darboux.c` :
   `float max_terrain(const mnt *restrict m)`
   Une réduction `max` est effectuée lors du calcul du maximum d'un terrain.
2. Ligne `40` dans `darboux.c` :
   `float *init_W(const mnt *restrict m)`
   Une boucle `for` imbriquée lors de l'initialisation d'un terrain.
3. Ligne `212` dans `darboux.c` :
   `mnt *darboux(const mnt *restrict m)`
   Une réduction `ou-binaire` est effectuée lors du calcul d'un terrain.

2.2 MPI

Le processeur logique est celui de rang zéro. Il s'occupera de la lecture en entrée, de l'initialisation, de l'écriture en sortie, de la comparaison avec la version de référence ainsi que le calcul des performances.

Nous avons débuté par la construction d'un `MPI_Datatype` et d'une structure `MPI` contenant les informations essentielles du terrain en entrée. Celle-ci sera partagée à chaque processeur logique utilisé via `MPI_Bcast`.

Ensuite, le terrain initial est réparti de manière équitable entre chaque processeur logique. Par exemple, avec trois processeurs logiques et un terrain initial composé de dix lignes, la répartition sera la suivante : 4-3-3. Cependant, ceci n'est pas suffisant car chaque processeur logique (sauf exception évidente) a besoin de la ligne précédente et/ou suivante. Ainsi, en reprenant l'exemple précédent, la répartition sera la suivante : 5-5-4.

Chaque processeur logique va donc, via `MPI_Scatterv`, recevoir sa fraction et y appliquer l'algorithme de Darboux. Or, appliquer cet algorithme pose problème car, bien qu'ils aient une répartition 5-5-4, ils ne calculent que pour une répartition 4-3-3. En somme, chaque processeur logique dépend de ceux qui l'entourent. Pour y remédier, à chaque itération, chacun enverra/recevra les lignes nécessaires (première et/ou dernière) via `MPI_Send/MPI_Recv`.

Afin de détecter la fin de l'algorithme, chaque processeur logique effectuera un `MPI_Allreduce` sur un booléen. Ainsi, si le booléen est à `false` pour chacun des processeurs, alors l'exécution de l'algorithme est terminée.

Enfin, un `MPI_Gatherv` est effectué pour les assembler en un terrain final.

Performances

3

3.1 OpenMP et MPI

3.1.1 Graphiques

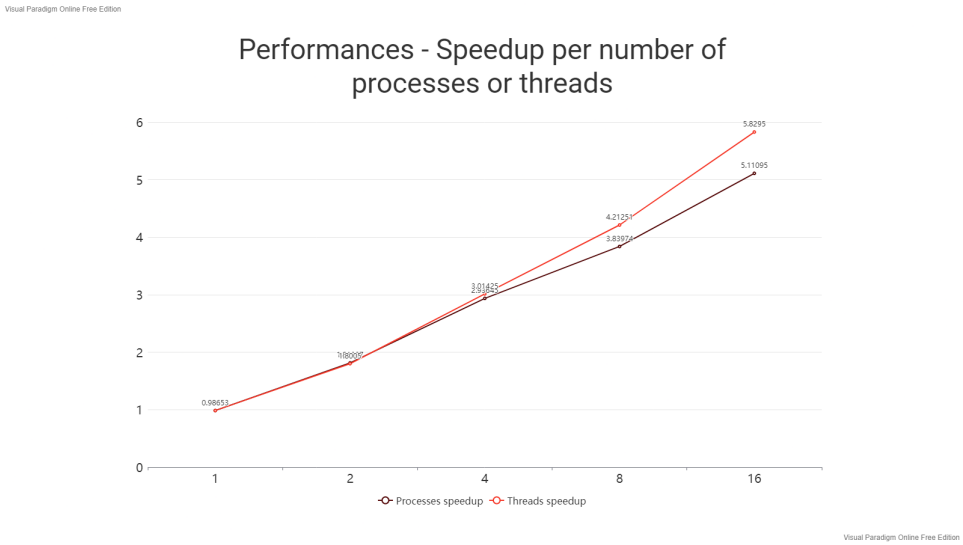


Figure 3.1 – Obtenu avec un processeur AMD Ryzen 7 2700X.

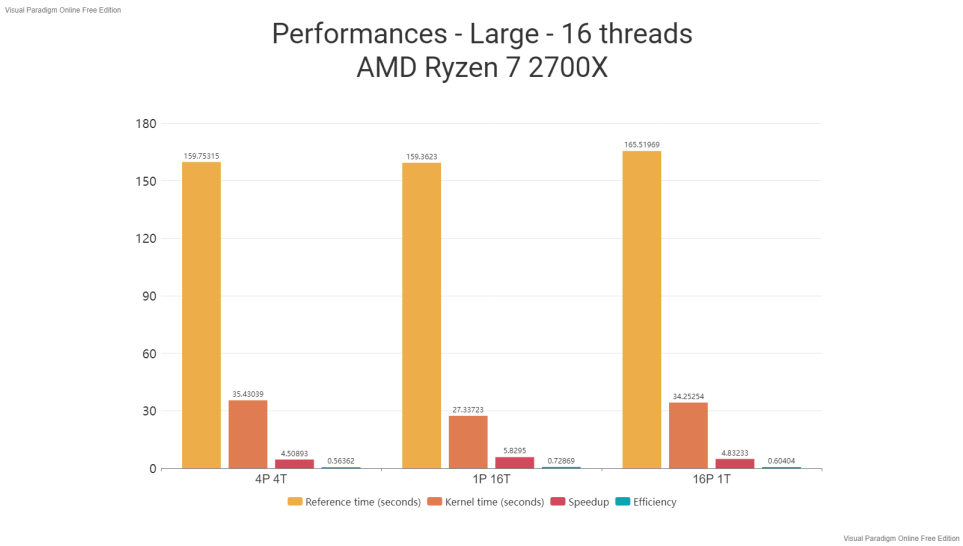


Figure 3.2 – Obtenu avec un processeur AMD Ryzen 7 2700X.

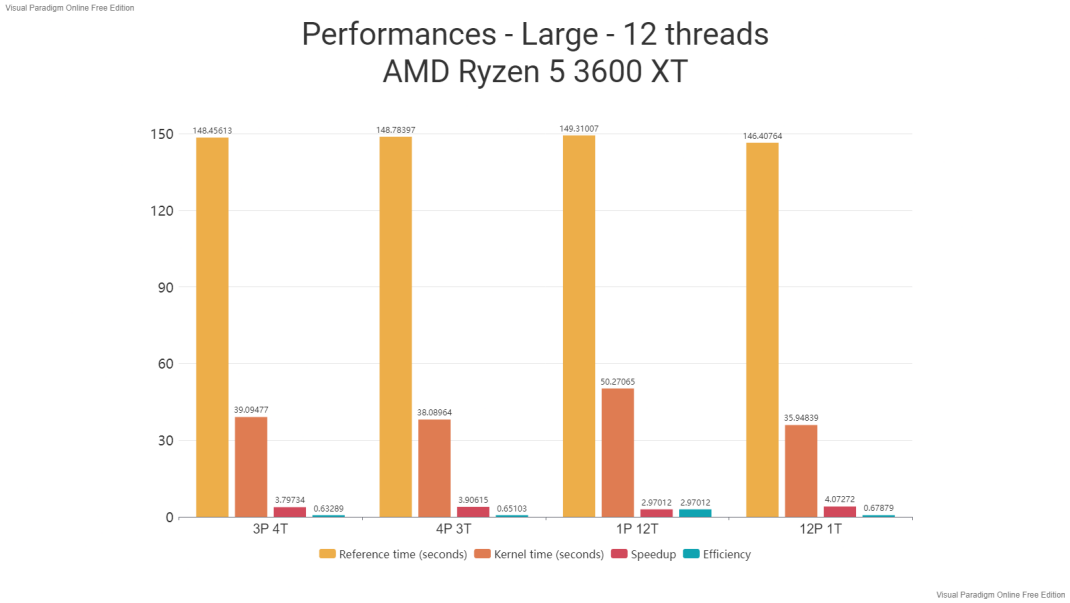


Figure 3.3 – Obtenu avec un processeur AMD Ryzen 5 3600XT.

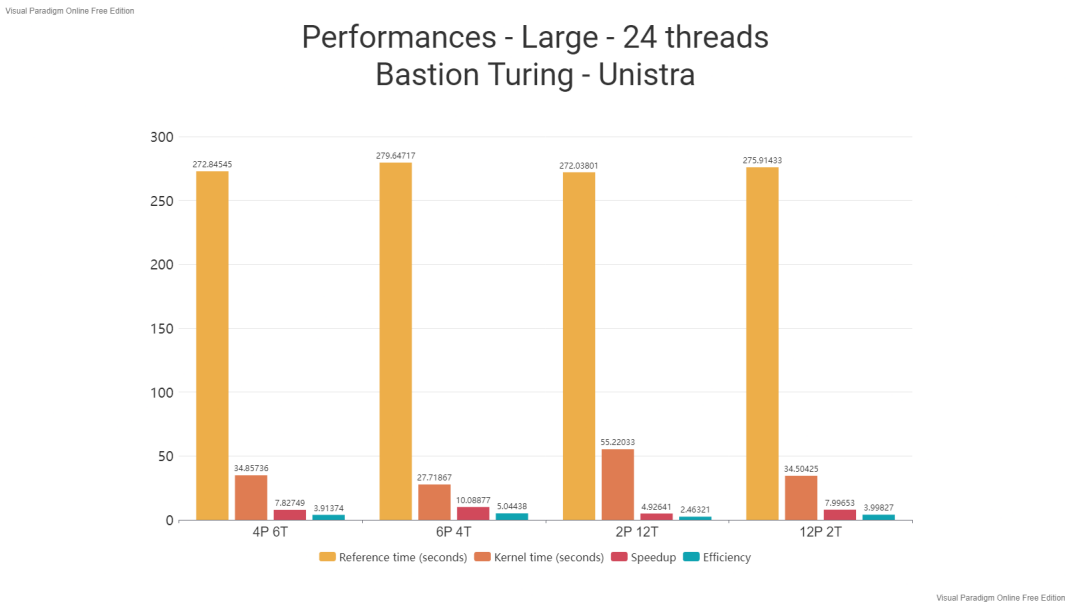


Figure 3.4 – Obtenu sur les VMs du bastion de l'université.

3.1.2 Analyse

La figure 3.1 montre l'évolution du speedup pour un nombre défini de processus, respectivement threads, en fonction du nombre de processeurs logiques, respectivement threads, utilisés lors de cette étude pouvant aller jusqu'à seize sur une machine possédant un processeur AMD Ryzen 7 2700X. En combinant cela avec la figure 3.2, ces graphiques suggèrent qu'à partir du nombre de huit, l'utilisation d'`OpenMP` serait plus intéressante que celle de `MPI`, mais reste équivalente avant ce nombre.

Pourtant, les figures 3.3 et 3.4 indiquent le total opposé, l'utilisation de `MPI` serait visiblement plus intéressante ici.

Nous en concluons que les performances sont naturellement liées à l'architecture de chaque processeur. Ainsi, `MPI` peut parfois être plus intéressant qu'`OpenMP` ou inversement. Cependant, nous remarquons également qu'une utilisation simultanée permet souvent de bien meilleures performances en moyenne, ce qui peut s'avérer être un point particulièrement attractif dans le cas d'une exécution sur une architecture inconnue.

3.2 Taille des données en entrée

3.2.1 Graphiques

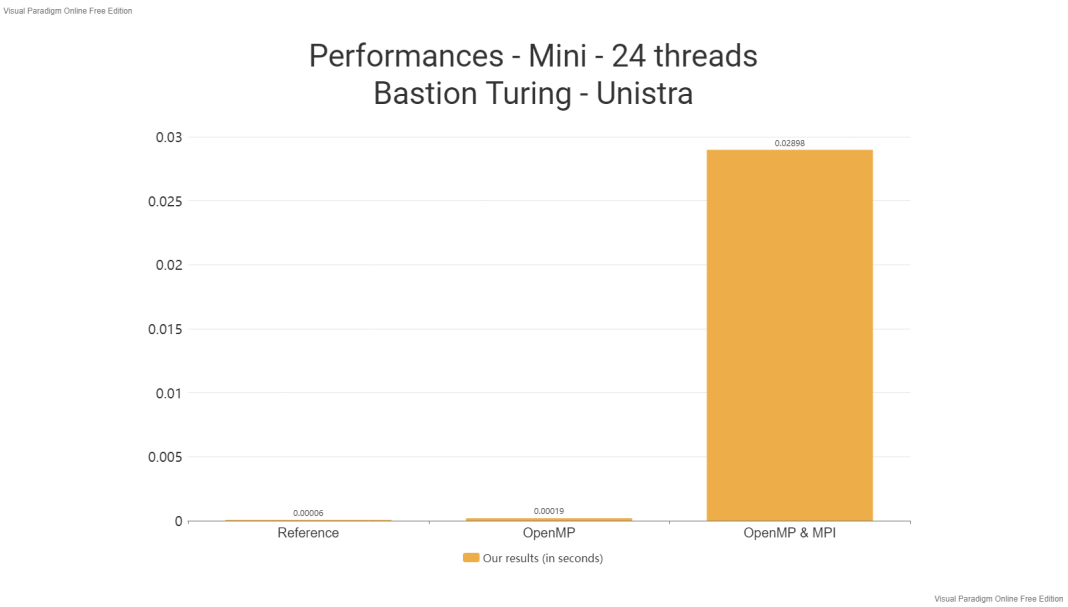


Figure 3.5 – Performances de l'input `mini.mnt` sur le bastion de l'université.

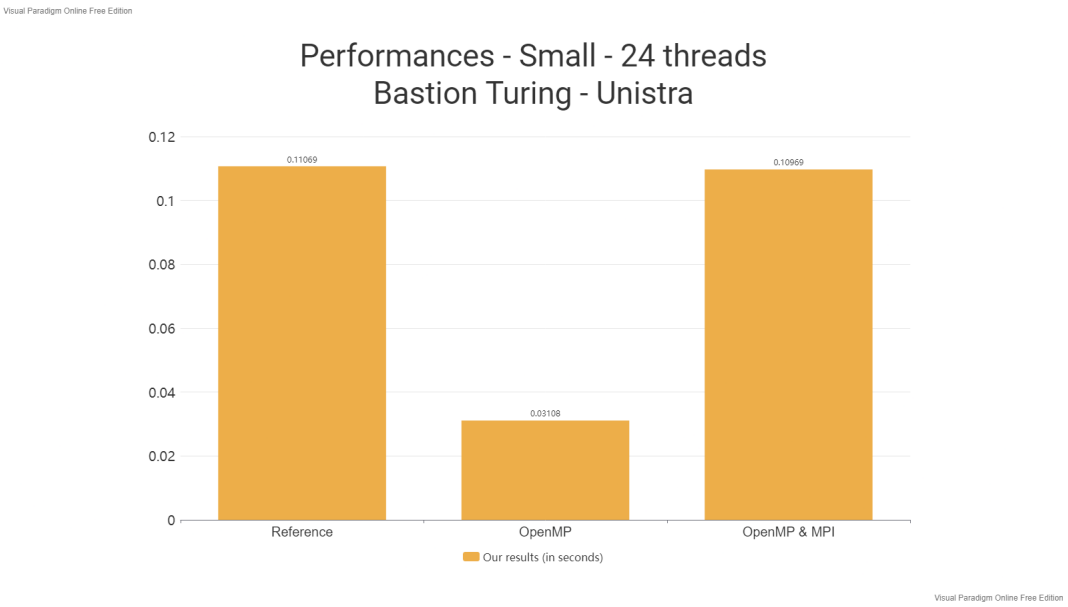


Figure 3.6 – Performances de l'input `small.mnt` sur le bastion de l'université.

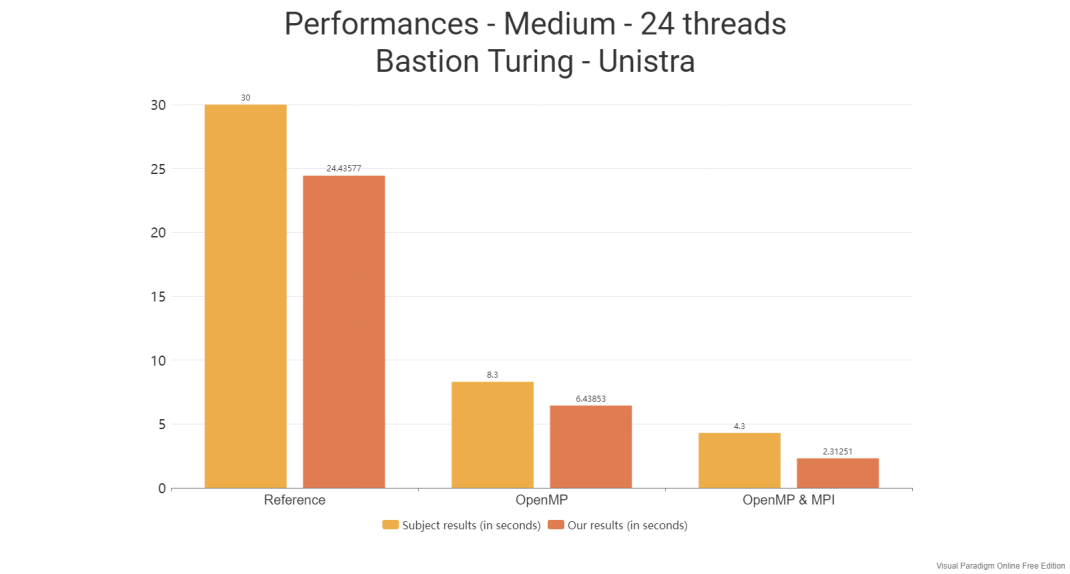


Figure 3.7 – Performances de l'input `medium.mnt` sur le bastion de l'université.

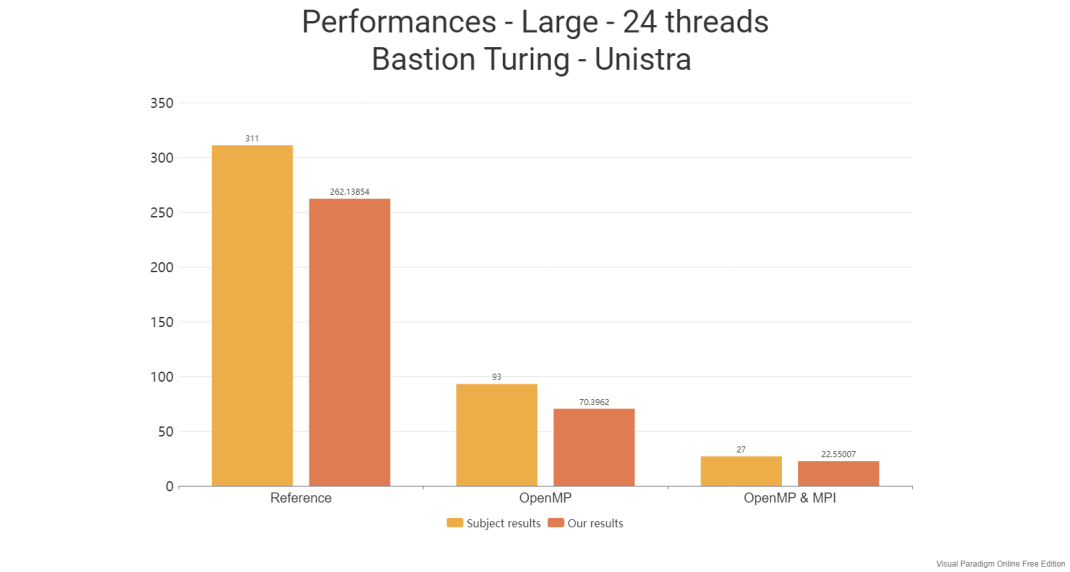


Figure 3.8 – Performances de l'input `large.mnt` sur le bastion de l'université.

3.2.2 Analyse

L'analyse de la figure 3.5 montre que parélliser n'est pas intéressant dans le cas où les données sont de très petite taille. On remarque que l'utilisation seule d'`OpenMP` demande environ trois fois plus de temps et que, combiné à `MPI`, cette exécution peut prendre jusqu'à 483 fois plus de temps.

En revanche, la figure 3.6 indique que paralléliser peut devenir intéressant si les données sont de petite taille. En effet, bien que le temps d'une exécution en séquentiel soit semblable au temps d'exécution avec `OpenMP` et `MPI`, une exécution à base d'`OpenMP` uniquement est environ trois plus rapide. Cependant, comme vu lors de la comparaison entre `OpenMP` et `MPI`, ceci est notamment dépendant de l'architecture.

Or, si les données en entrée sont plutôt de taille moyenne ou grande, alors l'utilisation d'`OpenMP` et `MPI` devient réellement intéressante. Cette exécution est environ trois plus rapide qu'une exécution à base d'`OpenMP` uniquement, elle même étant quatre fois plus rapide que sa version séquentielle.

On remarque naturellement que l'utilisation de `MPI` sur des données de petite taille n'est pas intéressant, ce qui est notamment du à l'attente provoquée par les transmissions via le réseau entre processus. Ce qui est l'opposé de `OpenMP`, qui lui est supposé être performant en toutes circonstances, hormis pour des boucles avec un faible nombre d'itérations.

Ainsi, nous en concluons que plus les données d'entrée sont petites, plus la parallélisation peut devenir un obstacle. Nous estimons qu'à partir du moment où les données en entrée ne sont pas de très petite taille, il devient possible de paralléliser. Cependant, c'est à partir du moment où les données sont au moins de taille moyenne que paralléliser devient intéressant.

4 Conclusion

Nous n'avons pas rencontré de difficultés particulières au cours de ce projet, hormis peut-être la répartition du terrain initial via l'utilisation de `MPI_Scatterv`.

Dans l'objectif d'une amélioration future, nous pourrions effectuer les échanges de lignes `MPI` simultanément à une partie du calcul. Ceci avait déjà été tenté d'être implémenté (voir commit), mais nous l'avons par la suite ôté au cours du développement. Nous avons maintenu notre décision car nous avons estimé nos performances comme étant suffisamment bonnes comparées à celles attendues et mentionnées dans le sujet.

Concernant les performances, nous en concluons que plus la taille les données en entrée est importante, plus la parallélisation permet d'obtenir de meilleures performances. En revanche, nous déconseillons la parallélisation sur des données de très petite taille (les performances seraient moins bonnes) ou de petite taille (l'architecture pourrait influer positivement ou négativement sur les performances, ce qui constitue un risque).