Projet d'HPC

sana.belmehdi, hajar.laktaoui, saida.elouardi

July 2025

1 Introduction

Ce projet porte sur la simulation d'une rupture de barrage en 1D à l'aide des équations de Saint-Venant. Le code de base, écrit en C avec le schéma de Rusanov, est parallélisé via OpenMP (partage de boucles) et MPI (décomposition de domaine). L'objectif est de comparer les performances et la validité des versions parallèles face à la version séquentielle et à la solution exacte.

2 Analyse du code séquentiel

Le programme C simule la rupture d'un barrage en 1D à l'aide des équations de Saint-Venant, un système utilisé pour modéliser les écoulements peu profonds. Il repose sur une discrétisation explicite des domaines spatial et temporel, et utilise le schéma de Rusanov pour approximer les flux numériques entre les cellules.

Le code initialise une discontinuité de hauteur d'eau représentant une rupture de barrage : niveau élevé à gauche, bas à droite, avec une vitesse initialement nulle. Une grille uniforme en 1D est utilisée pour discrétiser l'espace. À chaque pas de temps, le programme calcule la vitesse et la célérité, ajuste le pas de temps, évalue les flux avec le schéma de Rusanov, met à jour les variables conservées, puis applique des conditions aux bords. Les résultats finaux (position, hauteur, vitesse) sont enregistrés dans output.dat. Le temps d'exécution mesuré de cette version séquentielle est de **179,17s**.

3 Implémentation parallèle

L'exécution du code séquentiel est relativement longue (179,17s), en raison du grand nombre de points de grille traités à chaque pas de temps. Pour réduire ce temps de calcul, il est judicieux d'opter pour des approches parallèles.

3.1 Parallélisation avec OpenMP

OpenMP est une API simple et efficace pour exploiter le parallélisme sur les architectures à mémoire partagée. Dans notre projet, il a été utilisé pour paralléliser les boucles les plus coûteuses du code à l'aide de la directive **#pragma omp parallel for**, notamment celles liées au calcul de la vitesse, de la célérité, des flux et de la mise à jour des variables conservées.

3.2 Parallélisation avec MPI

MPI permet de distribuer le calcul sur plusieurs processus, ce qui est particulièrement adapté aux architectures distribuées. Dans notre projet, le domaine est divisé en blocs, chacun traité par un processus. Les échanges aux frontières se font via **MPI_Sendrecv** avec des zones fantômes, qui permet de combiner l'envoi et la réception de messages en une seule opération, ce qui facilite la synchronisation et évite les blocages dans les programmes parallèles., et le pas de temps est synchronisé avec **MPI_Allreduce**. Les résultats sont ensuite rassemblés avec **MPI_Gather**. Cette approche améliore la performance tout en assurant la cohérence de la solution par rapport à la version séquentielle.

4 Validation et comparaison

4.1 Évaluation des performances d'OpenMP

Pour analyser l'impact du parallélisme sur le code, nous avons réalisé une étude de strong scaling en mesurant le speedup et l'efficacité du programme, tout en faisant varier le nombre de threads à taille de problème constante.

- **SpeedUp** : l'étude met en évidence une bonne accélération du programme jusqu'à 4 threads, avec un speedup maximal d'environ 3,1×. Cela montre que notre code utilise bien le parallélisme, les threads travaillent en parallèle sans se gêner. Toutefois, au-delà, les performances diminuent, en raison de : le système passe plus de temps à gérer les threads (création, synchronisation, ordonnancement), Tous les threads accèdent à la même mémoire

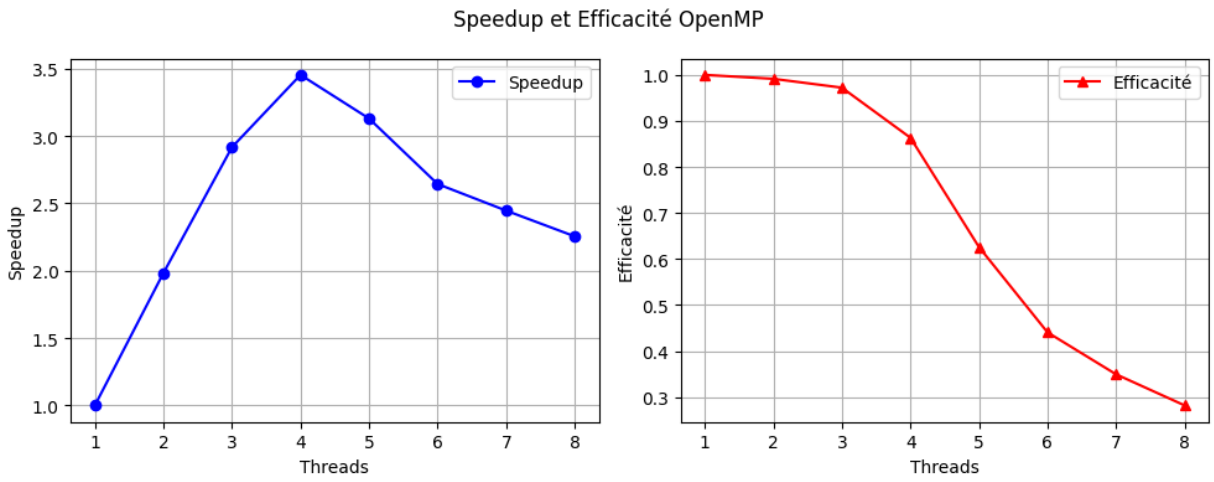

Figure 1: OpenMP Speedup et efficacité

partagée, ce qui provoque des conflits dans le cache et une saturation de la mémoire. Ainsi, les threads passent plus de temps à attendre qu'à calculer.

- **L'efficacité** : est parfaite jusqu'à 2 threads, puis chute progressivement jusqu'à 30% avec 8 threads. C'est logique : vu que l'efficacité est : Speedup/Nombre de threads. Donc, si on augmente les threads tant que le speedup n'augmente plus, l'efficacité diminue naturellement.

4.2 Évaluation des performances de MPI

Pour évaluer les performances de la version MPI, nous avons mesuré à la fois la **Weak scaling** et la **Strong scaling**.

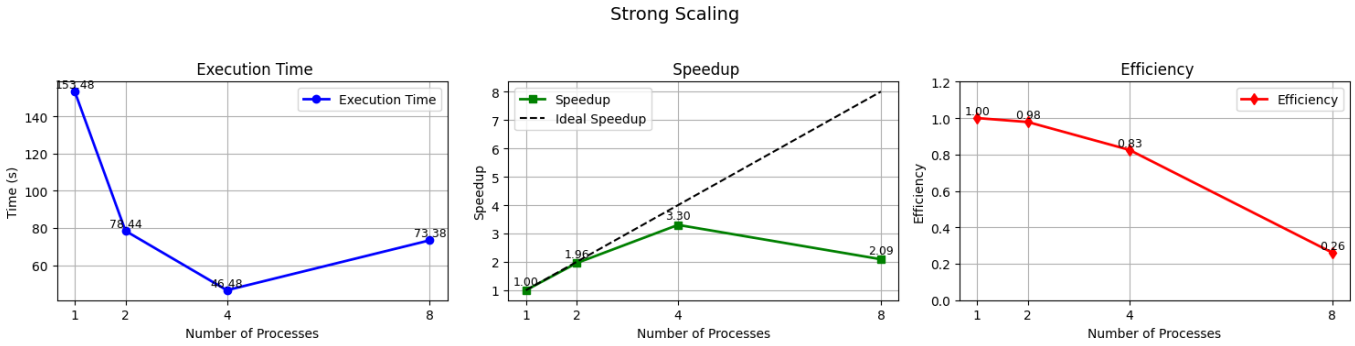

Figure 2: MPI Strong Scaling

Strong scaling : la figure ci-dessus présente trois sous-graphes illustrant l'analyse de strong scaling de l'implémentation MPI.

- **Le temps d'exécution** : le premier graphe montre que le temps d'exécution diminue significativement de 153,48 s à 46,48 s en passant de 1 à 4 processus, ce qui traduit un bon gain de performance initial avec une La figure ci-dessus présente trois sous-graphes illustrant l'analyse de strong scaling de l'implémentation MPI. En revanche, lorsque l'on augmente encore le nombre de processus plus de 4, le temps remonte. Ce qui signifie que les coûts de communication entre les processus et les Deadlocks did.loks (blocages) lorsque Deux ou plusieurs processus attendent indéfiniment que l'autre envoie ou reçoive un message, bloquant ainsi l'exécution ralentissent l'exécution.

- **SpeedUp et L'Efficacité** : le graphe central, représentant le speedup, montre une amélioration jusqu'à 3,30 à 4 processus, avant de redescendre à 2,09 à 8 processus, s'éloignant de la courbe idéale, tant que le dernier graphe reste très élevée jusqu'à 4 processus. Ensuite, il chute fortement, ce qui signifie que le programme devient moins efficace au nivau d'utilisation de ressources, et à cause des surcoûts liés à la communication MPI.

Weak Scaling : la figure, composée de trois sous-graphes, illustre les performances de l'implémentation MPI pour weak scaling, où le nombre de processeurs augmente proportionnellement à la taille du problème.

- **Le temps d'exécution** : le premier graphe montre une augmentation significative du temps d'exécution avec le nombre de processus, passant de 153,2 s à 4320,6 s entre 1 et 8 processus, ce qui traduit une surcharge importante liée aux communications.

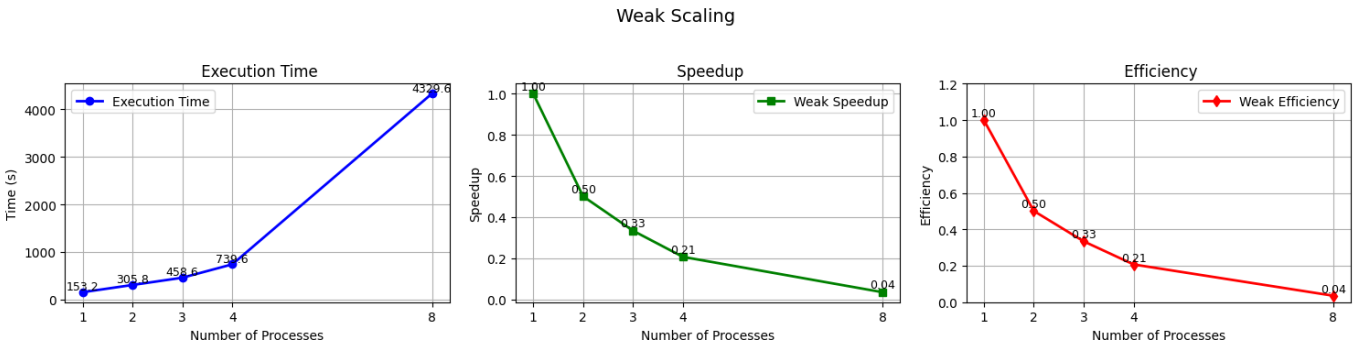

Figure 3: MPI Weak scaling

- **SpeedUp et L'efficacité** : le deuxième graphe révèle une chute marquée du speedup, atteignant seulement 0,04 à 8 processus, contrairement à la croissance attendue en weak scaling. Pour le troisième graphe met en évidence une forte baisse de l'efficacité, passant de 1,0 à 0,04, ce qui témoigne d'une très faible scalabilité. Ces résultats indiquent que le parallélisme est mal exploité dans cette configuration, en raison de communications MPI trop coûteuses et d'un volume de calcul insuffisant par processus.

4.3 Comparaison des solutions numériques et exacte

Pour valider la justesse de notre implémentation, nous comparons les solutions numériques (séquentielle et parallèle OpenMP et MPI) à la solution exacte à l'instant $t = 3{,}0\,s$.

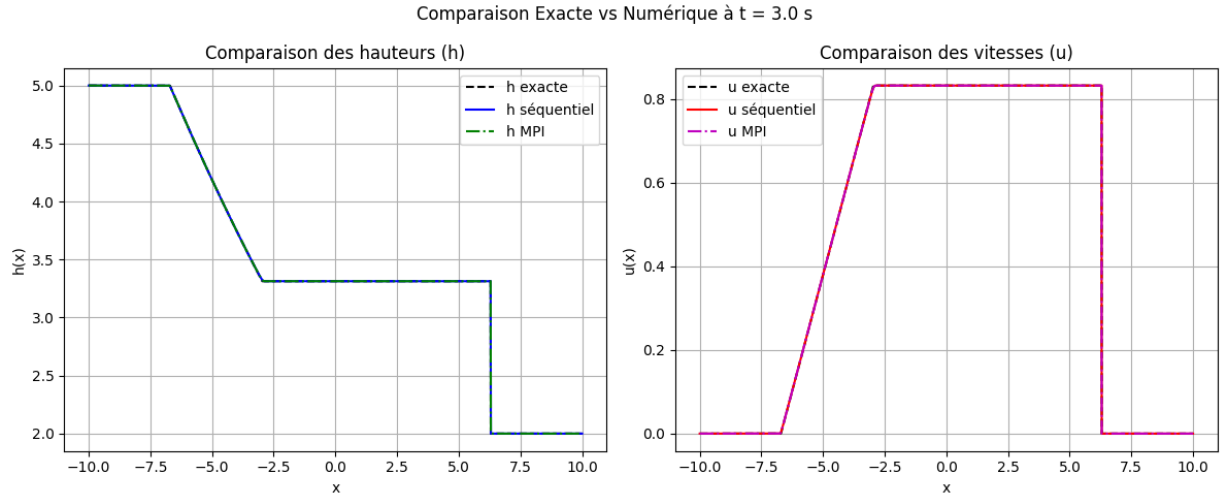

Figure 4: OpenMP strong Scaling

La figure présente une comparaison des hauteurs $h(x)$ et des vitesses $u(x)$ obtenues avec la solution exacte, la version séquentielle et la version parallèle MPI à l'instant $t = 3{,}0\,s$. Les courbes issues des trois approches sont quasiment superposées, tant pour les hauteurs que pour les vitesses, ce qui traduit une excellente précision numérique. Les discontinuités (chocs et zones de raréfaction) sont bien capturées, et les profils de vitesse sont correctement restitués. Cette concordance confirme que la parallélisation MPI est correctement implémentée, qu'elle préserve la justesse du schéma de Rusanov, et qu'elle ne compromet pas la qualité des résultats par rapport à la version séquentielle ou à la solution de référence.

5 Conclusion

La résolution du problème de dam break a permis de mettre en évidence les avantages des techniques de parallélisation pour accélérer les calculs numériques. En utilisant des méthodes comme OpenMP et MPI, des gains de temps significatifs peuvent être obtenus, à condition que la parallélisation soit correctement paramétrée, notamment en ce qui concerne le nombre de threads ou de processeurs utilisés.

Cependant, la parallélisation n'est pas toujours synonyme d'efficacité. Son impact dépend fortement de la nature du problème traité. Si elle est mal implémentée, elle peut au contraire allonger les temps de calcul, en particulier à cause des échanges d'informations entre processus qui génèrent une surcharge.

Il est donc essentiel d'adapter soigneusement la stratégie de parallélisation aux caractéristiques du problème et aux ressources disponibles, afin de garantir une amélioration réelle des performances.