fig

Projet parallèle :

Résolution du Dam Break 1D avec les équations de Saint-Venant (OpenMP & MPI)

Réalisé par :
Guerbouzi Firdawse

Encadré par :
Professeur Imad KISSAMI

July 2025

Table des matières

1 Introduction

La rupture de barrage est un scénario classique en simulation numérique de fluides, modélisé par les équations de Saint-Venant en une dimension. Ce phénomène présente une discontinuité initiale qui génère des ondes de choc et d'expansion, ce qui en fait un excellent cas test pour évaluer la stabilité et la précision de schémas numériques. L'objectif de ce projet est de paralléliser un code `C` séquentiel basé sur le schéma de Rusanov, afin d'en améliorer les performances. Deux approches complémentaires ont été mises en œuvre :

— **OpenMP** pour le parallélisme mémoire partagée, en exploitant les boucles parallélisables du code ;

— **MPI** pour le parallélisme distribué, en découpant le domaine et en gérant explicitement les échanges de données entre processus.

Le projet vise à comparer ces deux stratégies en termes de scalabilité (*strong* et *weak scaling*), de bande passante mémoire, et de fidélité des résultats. L'ensemble du travail s'appuie sur une méthode simple mais rigoureuse : adapter le code sans modifier la logique physique, mesurer, comparer, et analyser.

2 Méthodologie numérique

2.1 Schéma de Rusanov

Le schéma utilisé pour la résolution des équations de Saint-Venant est celui de Rusanov, également appelé schéma de Lax-Friedrichs local. Il s'agit d'un schéma de type volumes finis, explicite, de première ordre, particulièrement adapté aux problèmes hyperboliques avec discontinuités. À chaque pas de temps, le flux numérique entre deux mailles est calculé selon la formule suivante :

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left(F(U_i) + F(U_{i+1})\right) - \frac{1}{2} s_{\max}(U_{i+1} - U_i)$$

où $U = (h, hu)$ est le vecteur des variables conservées (hauteur et quantité de mouvement), $F(U)$ le flux physique, et $s_{\max}$ la vitesse maximale locale estimée à partir des vitesses et profondeurs des deux mailles adjacentes.

2.2 Structure du code séquentiel

Le code de base est écrit en langage `C` et suit une structure simple en plusieurs étapes :

1. Création du maillage 1D avec un espacement uniforme.
2. Initialisation des variables : hauteur $h(x)$ et vitesse $u(x)$, selon la position initiale par rapport à la discontinuité.
3. Calcul des variables conservées : $W = (h, hu)$.
4. Boucle temporelle principale :
   — Calcul de la vitesse locale et du pas de temps adapté (condition CFL) ;

— Calcul du flux numérique par le schéma de Rusanov ;
— Mise à jour explicite des variables conservées.

5. Écriture des résultats finaux dans un fichier de sortie.

Cette structure claire et modulaire a facilité l'ajout de directives de parallélisation avec OpenMP, ainsi que la division du domaine pour l'approche MPI.

3 Version OpenMP

Cette version vise à exploiter le parallélisme mémoire partagée à l'aide d'OpenMP, afin d'accélérer l'exécution du code séquentiel sur machine multicœur.

3.1 Directives de parallélisation

La parallélisation a été réalisée à l'aide de la directive `#pragma omp parallel for`, appliquée aux boucles indépendantes du code. Cela permet aux threads de traiter différentes portions des tableaux simultanément, réduisant ainsi le temps de calcul.

Les principales zones parallélisées sont :

— **Génération du maillage** : le tableau des positions peut être calculé indépendamment par chaque thread.

```
#pragma omp parallel for
for (int i = 0; i < nx; ++i)
    x[i] = XLEFT + i * dx;
```

— **Initialisation des conditions** : chaque cellule est initialisée en fonction de sa position sans dépendance aux autres.

```
#pragma omp parallel for
for (int i = 0; i < nc; ++i) {
    h[i] = (x[i] < XM) ? HLEFT : HRIGHT;
    u[i] = 0.0;
}
```

— **Calcul du flux de Rusanov** : les flux peuvent être calculés indépendamment car ils dépendent uniquement des cellules adjacentes.

```
#pragma omp parallel for
for (int i = 0; i < nc - 1; ++i) {
    // Calcul du flux[i]
}
```

— **Mise à jour des variables** : les cellules peuvent être mises à jour en parallèle (hors bords).

```
#pragma omp parallel for
for (int i = 1; i < nc - 1; ++i) {
    Wn[0][i] = ...;
```

```
        Wn[1][i] = ...;
    }
```

Gestion des accès concurrents : Le calcul du pas de temps (CFL) repose sur la vitesse maximale. Pour éviter les conflits entre threads, une réduction est utilisée :

```
#pragma omp parallel for reduction(max:max_speed)
for (int i = 0; i < nc; ++i) {
    double speed = fabs(u[i]) + sqrt(G * h[i]);
    if (speed > max_speed)
        max_speed = speed;
}
```

Enfin, les valeurs aux bords sont mises à jour de manière séquentielle pour éviter les écritures concurrentes.

3.2 Évaluation des performances

Les performances ont été évaluées selon deux axes : *scalabilité forte* et *scalabilité faible*.

Scalabilité forte (Strong Scaling)

La taille du domaine est fixée à `NX = 10001`, et le nombre de threads varie de 1 à 8. Pour chaque configuration, le programme est exécuté 10 fois et le temps moyen est enregistré. Les indicateurs utilisés sont :

— le **speedup** $S(p) = \frac{T_1}{T_p}$;

— l'**efficacité** $E(p) = \frac{S(p)}{p} \times 100\%$,

où T_p est le temps moyen avec p threads.

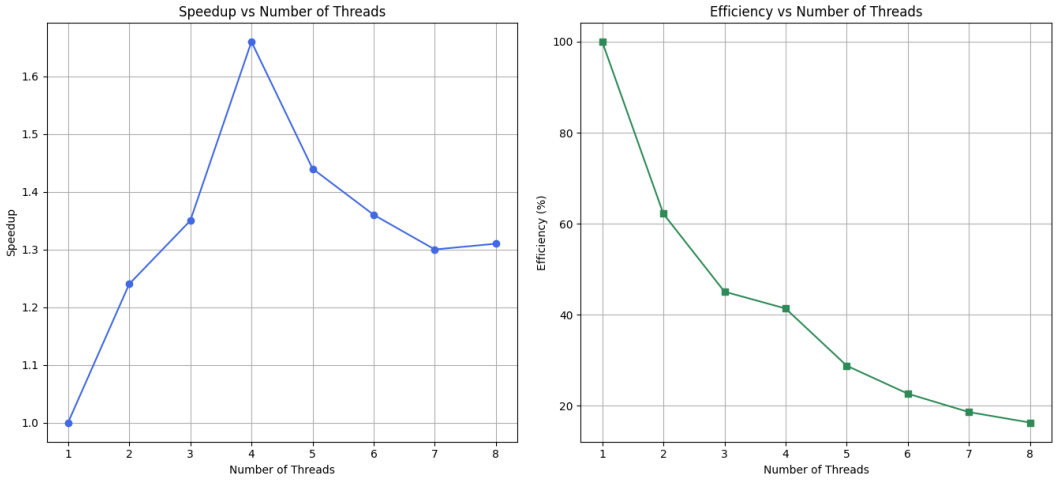

FIGURE 1 – Speedup et efficacité en fonction du nombre de threads (scalabilité forte)

La courbe de speedup montre une amélioration, avec un maximum autour de 1.68 obtenu à 4 threads. Dès 2 threads, l'efficacité chute rapidement et passe sous les 50 % à partir de 3 threads. Au-delà, les gains sont faibles voire nuls, et l'efficacité tombe à environ 15 % à 8 threads. Ces résultats suggèrent que La saturation de la mémoire partagée, les coûts de synchronisation et la faible quantité de calcul par thread sont probablement les principales raisons de cette perte d'efficacité. ainsi que la présence de portions séquentielles dans le code (par exemple, les bords). Ce comportement reste cohérent avec les limitations habituelles observées sur architectures à mémoire partagée.

Scalabilité faible (Weak Scaling)

Dans ce second test, la taille du domaine est proportionnelle au nombre de threads (`NX = 10001 × threads`), ce qui permet de garder une charge constante par thread.

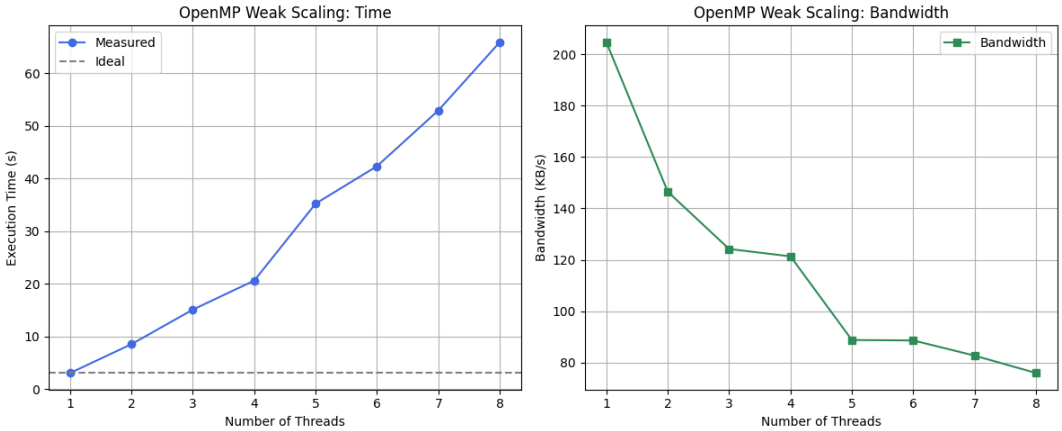

FIGURE 2 – Temps d'exécution et bande passante mémoire en fonction du nombre de threads (scalabilité faible)

La courbe de temps montre une augmentation linéaire avec le nombre de threads, ce qui n'est pas conforme au comportement idéal du weak scaling. Cela *peut s'expliquer* par une saturation progressive de la bande passante mémoire, confirmée par la seconde courbe. Cette dernière montre une chute rapide de la bande passante dès 2 threads, puis une stagnation à partir de 5 threads autour de 80–90 KB/s. Il est *probable que* la contention sur les accès mémoire partagés devienne un facteur limitant, même si la charge est bien répartie.

3.3 Vérification de la cohérence

La solution obtenue avec OpenMP a été comparée à la solution exacte du problème. Le graphique ci-dessous montre que les deux courbes sont quasi confondues, validant ainsi la justesse du résultat.

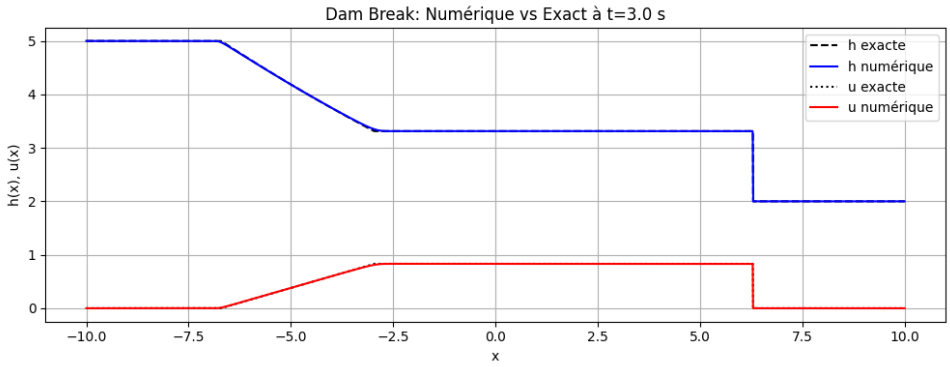

FIGURE 3 – Comparaison entre la solution parallèle et la solution exacte

4 Version MPI

La version MPI repose sur une décomposition 1D du domaine, permettant d'exécuter le code sur plusieurs processus en parallèle. Chaque processus est responsable d'un sous-domaine, et des échanges de données ont lieu aux interfaces afin de maintenir la continuité physique de la solution.

4.1 Directives et parallélisation

1. Découpage du domaine Le domaine global est réparti en sous-domaines de tailles presque égales, selon un découpage par blocs avec gestion du reste. Chaque processus travaille uniquement sur sa portion locale :

```
int base = nc_global / size;
int rest = nc_global % size;
int nc_local = base + (rank < rest ? 1 : 0);
int start_index = rank * base + min(rank, rest);
```

2. Gestion des cellules de communication Pour que chaque processus puisse calculer les flux aux interfaces, il a besoin des valeurs des cellules adjacentes des sous-domaines voisins. C'est pourquoi on ajoute deux cellules fantômes autour du domaine local : une à gauche et une à droite. Ces deux emplacements supplémentaires permettent de recevoir les valeurs de bord des processus voisins sans écraser les données internes.

L'allocation des tableaux est donc faite avec deux cases supplémentaires :

```
double *h = malloc((nc_local + 2) * sizeof(double));
double *u = malloc((nc_local + 2) * sizeof(double));
```

L'indice `h[1]` correspond à la première cellule réelle, tandis que `h[0]` et `h[nc_local+1]` sont réservés pour les valeurs reçues.

3. Gestion des bords (communications MPI) À chaque pas de temps, les cellules fantômes sont remplies à l'aide d'échanges avec les processus voisins. Ces échanges se font via `MPI_Sendrecv`, pour envoyer la bordure réelle et recevoir celle du voisin :

```
// change avec le voisin de gauche
if (rank > 0) {
    MPI_Sendrecv(&h[1], 1, MPI_DOUBLE, rank - 1, 0,
                 &h[0], 1, MPI_DOUBLE, rank - 1, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&u[1], 1, MPI_DOUBLE, rank - 1, 2,
                 &u[0], 1, MPI_DOUBLE, rank - 1, 3,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

// change avec le voisin de droite
if (rank < size - 1) {
    MPI_Sendrecv(&h[nc_local], 1, MPI_DOUBLE, rank + 1, 1,
                 &h[nc_local + 1], 1, MPI_DOUBLE, rank + 1, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&u[nc_local], 1, MPI_DOUBLE, rank + 1, 3,
                 &u[nc_local + 1], 1, MPI_DOUBLE, rank + 1, 2,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Ces échanges garantissent que chaque processus peut travailler localement tout en ayant une vue cohérente du domaine global.

4. Synchronisation du pas de temps Le pas de temps dépend de la vitesse maximale locale. Pour garantir une évolution temporelle cohérente sur tous les processus, une synchronisation est effectuée avec `MPI_Allreduce` :

```
MPI_Allreduce(&local_max, &global_max, 1, MPI_DOUBLE,
              MPI_MAX, MPI_COMM_WORLD);
double dt = CFL * dx / global_max;
```

5. Reconstruction de la solution globale À la fin de la simulation, les résultats locaux sont transmis au processus 0 à l'aide de `MPI_Gatherv` pour produire la solution complète sur l'ensemble du domaine.

4.2 Évaluation des performances

Scalabilité forte (Strong Scaling)

Dans ce test, la taille du domaine est fixée à `NX = 10001` et le nombre de processus varie de 1 à 8. Le temps d'exécution est mesuré avec `MPI_Wtime()`, puis comparé au temps séquentiel pour calculer le speedup et l'efficacité.

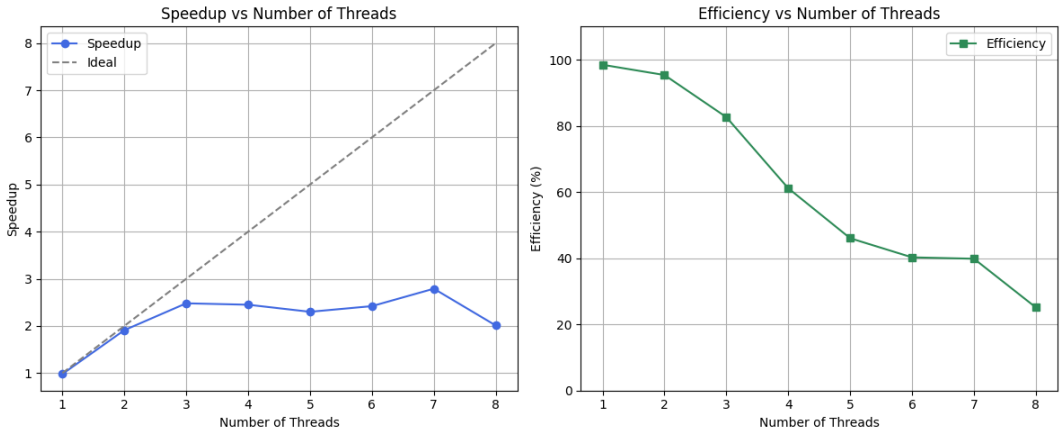

FIGURE 4 – Speedup et efficacité en fonction du nombre de processus (MPI – strong scaling)

On observe un speedup croissant jusqu'à 3 processus, atteignant environ 2.5, avec une efficacité supérieure à 80 %. Ensuite, la progression ralentit : le speedup plafonne entre 4 et 6 processus, avec un léger rebond à 7. À 8 processus, une chute brutale est visible, avec un speedup retombant à 2.0 et une efficacité autour de 25 %. Ce comportement peut indiquer que la surcharge de communication devient dominante. Il est également possible qu'à 8 processus, les blocs deviennent trop petits par rapport au coût fixe de coordination. Ce point de rupture illustre les limites du parallélisme distribué sur des problèmes de taille modeste.

Scalabilité faible (Weak Scaling)

Ici, la taille du domaine est proportionnelle au nombre de processus, maintenant une charge constante par processus. On mesure le temps d'exécution ainsi qu'une estimation de la bande passante effective.

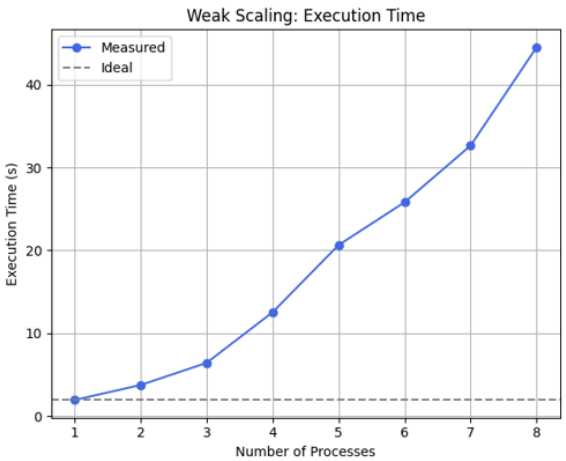

FIGURE 5 – Temps d'exécution(MPI – weak scaling)

La courbe montre une augmentation progressive du temps d'exécution avec le nombre de processus, ce qui s'éloigne du comportement idéal attendu en weak sca-

ling. Bien que la charge de calcul par processus soit constante, le temps global augmente, ce qui suggère que des surcoûts apparaissent avec la montée en parallélisme. Il est probable que ces surcoûts soient liés aux communications croissantes entre processus, notamment les échanges aux interfaces et la synchronisation du pas de temps. Cela montre que dans un environnement MPI, ce sont souvent les communications, plus que le calcul lui-même, qui limitent la performance lorsque l'on augmente le nombre de processus.

4.3 Vérification de la cohérence

La solution obtenue avec MPI a été comparée à celle du code séquentiel. Le graphique suivant montre une bonne superposition entre les deux profils, confirmant que la parallélisation n'altère pas la fidélité de la simulation.

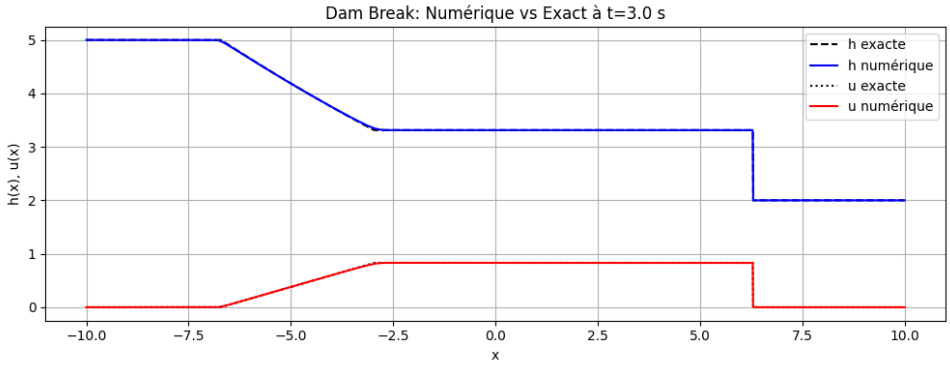

FIGURE 6 – Comparaison entre la solution MPI et la solution exacte

5 Conclusion

Ce travail a permis d'explorer concrètement la parallélisation d'un schéma numérique simple mais robuste sur deux architectures différentes : mémoire partagée (OpenMP) et mémoire distribuée (MPI). Les résultats obtenus montrent que si le parallélisme peut apporter des gains significatifs, il introduit aussi des contraintes propres à chaque modèle. En parallèle, la fidélité de la solution physique a été conservée, confirmant la validité des implémentations. Ce projet constitue ainsi une base solide pour des simulations plus complexes ou de plus grande échelle.