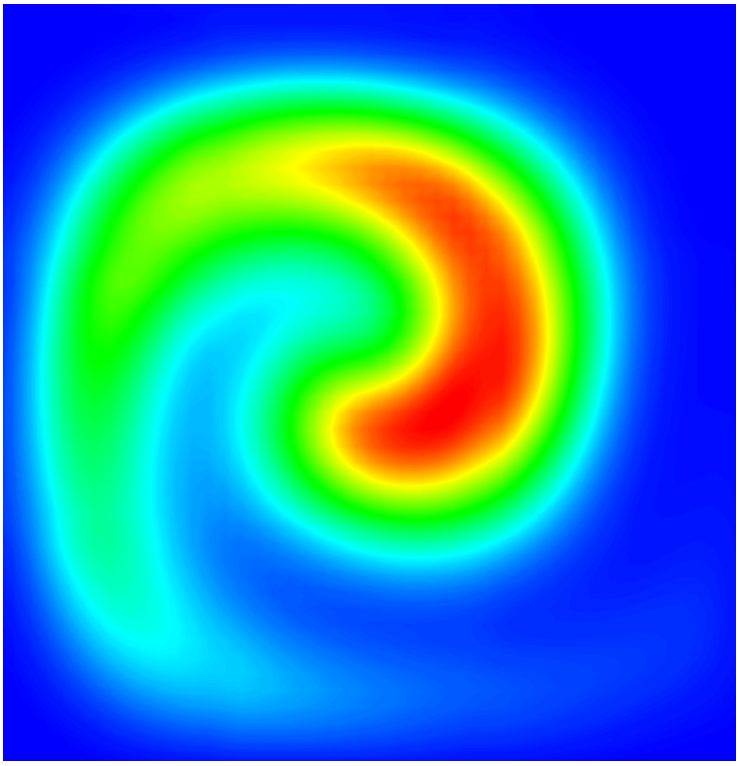
Projet de semetre
Prof. Florain Desmons
Semestre d' automne 2024-2025

Informatique et systèmes de communication ISC
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften



Example de simulation du transport d'espèce

Rapport du projet de semestre

Le transport d'éspèces

Marta Rende

5 juin 2024

Table des matières

1 - Introduction

Les objectifs du projet de semestre étaient de combiner la physique et l'informatique en réussissant à coder des simulations physiques. La simulation physique finale consiste à simuler le transport d'espèce qui étude l'évolution de la répartition spatiale d'une espèce dans le temps et dans l'espace.

Le deuxième objectif du projet est de pouvoir réaliser la simulation physique le plus rapidement possible. Ce point est très important car le problème en question implique la possibilité d'avoir une très grande variation dans sa complexité, ce qui pourrait conduire à des temps de calcul très élevés. Disposer d'un code permettant d'optimiser le temps de calcul est essentiel, il faut donc trouver un moyen de le paralléliser efficacement.

2 - Explication du problème physique

On va maintenant aborder le problème du point de vue de la physique. Pour déterminer comment l'espèce évolue dans le temps, on travaillera avec un maillage 2D dans lequel on trouvera nx x ny cellules. Plus nx et ny sont grands, plus la taille du problème sera importante et donc le calcul de l'espèce sera de plus en plus lent, mais l'augmentation de la taille permettra également d'obtenir une plus grande précision.

L'équation du transport d'espèce est la suivante:

$$\frac{\delta Y}{\delta t} - D(\Delta Y) + u(\nabla Y) + v(\nabla Y) = 0$$

ou

Y = variable du transport d'espèce

t = temps

D = coeff. de diffusion

u = vitesse de rotation sur l'axe des x

v = vitesse de rotation sur l'axe des y

On peut voir que l'équation est divisée en deux parties différentes, qui sont les parties qui composent la simulation des espèces. La première partie est celle de la diffusion et la seconde est celle de l'advection, qui consiste en un processus de transport du flux.

Notre objectif est de savoir comment l'espèce se comporte à l'instant de temps suivant, on peut donc transformer l'équation de la figure les équations suivantes:

$$\frac{Y^{n+1} - Y^n}{\Delta t} - D\Delta Y^n + u(\nabla Y) + v(\nabla Y) = 0$$

$$\Longleftrightarrow$$

$$Y^{n+1} = Y^n + \Delta t D \Delta Y^n - \Delta t u \nabla Y^n - \Delta t v \nabla Y^n$$

La dernière équation sera alors calculée pour chaque cellule de notre maillage. Elle représente comment calculer Y au moment suivant de manière explicite, cela signifie que Y au temps n+1 est calculé avec les valeurs de Y au temps n.

Pour notre projet, on a besoin d'une plus grande stabilité numérique, dont une méthode explicite ne peut pas nous offrir. On doit donc transformer notre équation sous forme implicite, ce qui permet à notre système de rester stable indépendamment du pas de temps de notre simulation.

L'équation transformée devient la suivante:

$$Y^{n+1} - \Delta t D \Delta Y^{n+1} = Y^n - \Delta t u \nabla Y^n - \Delta t v \nabla Y^n$$

Avec cette nouvelle équation on va donc calculer la partie de diffusion au temps n+1.

Notre équation peut donc être représentée par le système suivant:

$$Ax = b$$

ou

b = partie d'advection

x = $Y(n+1)$

A = partie de diffusion

A ce sera donc une matrice de taille (nx x ny)x(nx x ny) avec des coefficients extraient par rapport au calcul de ΔY^{n+1} .

3 - Approche informatique du problème

Dans cette section, on va aborder les choix effectués et la mise en œuvre d'un point de vue informatique. Globalement on a un programme qui permet de simuler le transport d'espèce. Pour visualiser la simulation on va produire des fichiers VTK.

3.1 - Où trouver le code

Le code peut être trouvé dans le repo github au lien suivant: https://github.com/MartaRende/SpeciesTransport.git . Il y aura également une copie du code séquentiel et parallèle dans le retour du projet. Dans le repo github, il y aura une branch « sequential » contenant le code séquentiel, tandis que sur la branch « main » il y aura la version parallèle du code.

3.2 - Outils utilisés

En ce qui concerne le choix des outils utilisé, le projet a été développé en C++. Ce choix a été fait en raison de la présence de librairies qui permettent une parallélisation efficace du code puisqu'il n'est pas nécessaire de passer par des API et parce que des approches en C++ ont déjà été abordées pendant les cours. Les librairies utilisées pour la parallélisation sont MPI (pour paralléliser sur CPU) et CUDA (pour paralléliser sur GPU). On va discuter ci-dessous des choix effectués pour chaque partie du code. Afin d'étudier les performances du code en parallèle, pour CUDA, nvprof (https://docs.nvidia.com/cuda/profiler-users-guide/#nvprof) a été utilisé. Visit (https://visit-dav.github.io/visit-website/index.html) a été utilisé pour afficher les résultats physiques. Finalement, calypso a été utilisé pour compiler et lancer le code, il a été essentiel pour avoir une performance du code plus élevée. Excel a été utilisé pour collecter des statistiques sur les performances du code et créer des graphiques. Pour gérer les erreurs CUDA, un fichier en c (common_includes.c) a été utilisé. Ce fichier est repris du cours de Data Computation.

3.3 - Explication du code

Le code de résolution de la simulation physique est divisé en plusieurs parties. Chacune de ces parties a d'abord été implémentée en séquentiel puis en parallèle.

Le code consiste en une initialisation de la simulation, qui se trouve dans le dossier « initialisation ». L'initialisation permet de définir un état de départ pour notre simulation. Elle a été parallelisé grace à l'aide de CUDA. Elle permet de initialiser les 6 différents espèces.

Après l'initialisation de la simulation, on a une partie qui gère le cœur de la simulation et qui est répétée plusieurs fois dans le temps. L'équation des espèces est résolue dans le dossier « solve ». Dans ce

dossier in va donc résoudre le système $Ax = b$. Ce processus de résolution peut à son tour être divisé en plusieurs parties.

Dans la première partie, la matrice A est remplie. Le problème de cette matrice est qu'elle est très grande (nx x ny) x (nx x ny) et elle occupe donc beaucoup de mémoire. Cela limite la taille du problème qu'on peut tester. Pour résoudre ça, des sparse matrix on été utilisées. Elles permettent d'enregistrer que les valeurs non zéros. La première étape consiste donc à remplir correctement la matrice en étant capable d'initialiser de bons indices de colonnes et de bons décalages de lignes. Ma matrice SparseMatrix se compose d'un tableau représentant les valeurs non nulles, d'un tableau représentant les indices de colonne qui correspondent au numéro de la colonne dans laquelle se trouvent les valeurs non nulles, et d'un tableau avec les décalages de ligne représentant le nombre de valeurs non nulles pour chaque ligne.

La deuxième étape de la résolution de l'équation consiste à résoudre la partie b du système Ax = b. b correspond à un tableau de taille nx x ny.

Dans la troisième partie, les valeurs précédemment calculées de A et b doivent être rassemblées pour trouver x soit Y^{n+1}. Trois méthodes ont été testées pour ce faire. La première était la méthode de l'inversion matricielle, c'est-à-dire $x = A^{-1}b$. Cependant, cette méthode s'est révélée très lente, avec une complexité algorithmique de O(n^3). Dans un deuxième temps, une approche itérative a été choisie, celle du gradient conjugué, qui s'est encore révélée très lent. Enfin, la méthode de Jacobi a été utilisée. Elle permet de trouver x à l'aide de la formule itérative suivante :

$$Yi(n+1) = \frac{1}{aii}\left(bi - \sum_{j=1, j\neq i}^{n} aij * xi^{k}\right)$$

Ces 3 différents morceaux ont été parallélisés sur le GPU grâce à cuda. Ils sont calculés et traités avec des kernel 2d. Au total, on a 4 kernel. Le premier est utilisé pour calculer les décalages de lignes de la matrice A, le second est utilisé pour remplir les valeurs non zéros de A et les indices de colonnes de A. Le troisième est utilisé pour calculer b et le quatrième est utilisé pour calculer x grâce à la méthode itérative de jacobi. La version séquentielle fonctionne de la même manière mais le remplissage de la matrice A se fait avec le type SparseMatrix de c++ alors que dans les kernels la matrice A est constituée de trois vecteurs distincts indiquant les valeurs, les décalages de lignes et les indices de colonnes.

Tous les kernel, utilisés sont en 2D. Ce choix a été fait parce que de nombreuses structures de données étaient à la base en 2D et il était, donc de manière subjective, plus facile de calculer les indices correspondants dans les kernels. Cela permet également de diviser le travail en blocs égaux. Une taille de bloc de 16x16 a été choisie, ce qui signifie que 256 threads peuvent être exécutés dans chaque bloc. Le nombre 16 a été choisi pour deux raisons. La première est dictée par le fait que notre gpu contient 32 threads dans un wrap, ce qui signifie que 32 threads exécutent la même instruction sur des données différentes en même temps. En plus, comme 16 est un multiple de 32, on a une utilisation maximale des wraps. Deuxièmement, le gpu disponible sur calypso a la capacité d'exécuter un maximum de 1024 threads par bloc. Cependant, il est parfois préférable de ne pas utiliser la capacité maximale du gpu car on risquerait de la saturer. 256 threads reste donc un bon compromis. Ensuite, la taille de la grille doit être calculée, c'est-à-dire le nombre de blocs nécessaires pour couvrir l'ensemble du domaine nx x ny. Ce calcul est effectué par excès. Étant donné que nx et ny ne seront pas toujours des multiples de 16, si on ne tiens pas compte de l'arrondi, on risque de perdre des parties du domaine qui sont importantes pour le résultat final.

Concernant l'écriture des fichier, MPI a été utilisée. Le processus commence avec l'écriture des headers. Le processus de rang 0 écrit l'en-tête du fichier et diffuse sa taille à tous les processus, établissant ainsi l'offset initial. Pour chaque section de données (espèces Yi, u, v), les offsets sont calculés de manière incrémentielle. Par exemple, pour les espèces, l'offset de chaque espèce est déterminé en ajoutant la

taille de l'en-tête spécifique à l'espèce, à l'offset précédent et après l'écriture de chaque espèce, l'offset global est mis à jour. Les offsets mis à jour sont diffusés à tous les processus via MPI_Bcast, assurant une connaissance partagée des positions d'écriture. Chaque processus utilise ces offsets synchronisés pour écrire ses données à la position correcte dans le fichier, en utilisant MPI_File_write_at.

Tous ces éléments sont réunis dans le programme principal. Le main est utilisé pour initialiser la simulation, en initialisant tous les devices nécessaires pour travailler sur le gpu de manière à ce que le moins d'allocations de mémoire possible soient effectuées. Ceci est important car les allocations de mémoire peuvent prendre beaucoup de temps et rendre notre code plus lent que la version séquentielle. Dans la partie principale, on calcule comment l'espèce évolue dans le temps en écrivant le résultat dans des fichiers vtk toutes les 100 itérations. Les fichiers vtk contiendront six variables différentes, une pour chaque espèce du logo ISC. En plus on a deux variables u et v pour les vitesses, nécessaires pour la partie advection. Chaque espèce aura son propre coefficient de diffusion. Afin de permettre une optimisation maximale, les copies de mémoire du device (GPU) vers l'host ne sont effectuées que lorsque des fichiers doivent être écrits, c'est-à-dire toutes les 100 itérations.

3.4 - Comment lancer le code

Cette section permettra de comprendre comment lancer le code et quels sont les prérequis nécessaires pour le faire.

3.4.1 - Prérequis

On va maintenant aborder les prérequis nécessaires pour lancer le code.

3.4.1.1 - Version séquentiel

Pour lancer le code en séquentiel, il est nécessaire d'avoir installé le compilateur G++ (https://code.visualstudio.com/docs/cpp/config-mingw) pour lancer un programme en c++. Il est également conseillé d'installer Visit(https://visit-dav.github.io/visit-website/index.html), qui permettra d'afficher le résultat de la simulation physique.

3.4.1.1.1 - Comment compiler et exécuter le code

Sur cette version il y aura un MakeFile qui permet de lancer le programme en locale en produisant dans le folder « ouptut » les fichiers nécessaires pour exécuter la simulation sur Visit.

Il suffira juste de écrire dans le terminale à la racine du projet « make » pour compiler le programme et puis « ./run_speciesTransport » pour exécuter le code.

3.4.1.2 - Version parallèle

Pour la version parallèle, il faut installer nvcc (https://docs.nvidia.com/cuda/cuda-installation-guide-linux/), le compilateur NVIDIA CUDA, qui compilera les fichiers nécessitant la parallélisation du GPU, ainsi que mpic++ (https://docs.open-mpi.org/en/v5.0.x/installing-open-mpi/downloading.html) pour compiler les fichiers nécessitant mpi et nvcc. Avant cela, il faut s'assurer que le GPU de l'ordinateur est compatible. Si ce n'est pas le cas, il faut trouver un device qui a cette compatibilité.

3.4.1.2.1 - Comment compiler et exécuter le code

Si toutes les conditions mentionnées ci-dessus sont remplies, la compilation peut être effectuée par le MakeFile du projet. Il suffit d'être à la racine du projet pour pouvoir le lancer.

Le MakeFile a deux options de lancement :

- main : Lance le programme principal, permettant de sortir le temps pris par le programme et les fichiers .vtk qui montrent l'évolution de la simulation dans le temps.

- test : permet de lancer les tests unitaires réalisés pour vérifier que les fonctions en parallèle produisent les résultats attendus.

Ainsi, pour lancer le programme principal, il suffit de faire make main, et pour lancer les unitTests, il suffit de faire make test. Pour l'exécuter, la procédure est la même que pour la version en séquentiel.

Si c'est pas possible compiler le programme localement, il faut utiliser un serveur à distance. Pour le lancer, il fuadra faire un un fichier qui, une fois lancé, permettra de compiler et d'exécuter le programme sur le serveur distant. Dans le cadre du projet, le serveur calypso a été utilisé, auquel il est possible de se connecter à distance via ssh. Voici un exemple de fichier .sh qui peut être utilisé pour exécuter le programme sur le serveur. Il a la structure suivante :

```
#!/bin/bash
sshqfunc() { echo "bash -c $(printf "%q" "$(declare -f "$@"); $1 \"\$@\"")"; };
remote() {
    # compile project
    cd   /path/of/project/in/remote/server/$1
    make main

  ./run_speciesTransport
}

echo "Copying directory"
# copy project in destination path
name=$(date +'%H%M%S')

ssh user@ip -t "mkdir -p /path/of/project/in/remote/server/$name/output "
rsync -az  ./solve/ user@ip:/path/of/project/in/remote/server/$name/solve &
rsync -az  ./write/ user@ip:/path/of/project/in/remote/server/$name/write &
rsync   -az      ./initialization/   user@ip:/path/of/project/in/remote/server/$name/
initialization &
rsync -az  ./unitTests/ user@ip:/path/of/project/in/remote/server/$name/unitTests &
rsync  ./common_includes.c user@ip:/path/of/project/in/remote/server/$name &
rsync  ./main.cu user@ip:/path/of/project/in/remote/server/$name
rsync  ./Makefile user@ip:/path/of/project/in/remote/server/$name

echo "Starting compilation"
ssh user@ip "$(sshqfunc remote)" -- $name
```

Listing 1 - Exemple de file pour la connection ssh à un server remote

La première partie permet de faire une liste d'actions à effectuer sur le serveur distant puis de compiler (make) et d'exécuter (./run_speciesTransport), la seconde permet de copier les fichiers nécessaires à l'exécution du programme. Pour la partie de simulation est possible aussi exécuter le programme avec MPI, donc au lieu de exécuter avec ./run_speciesTransport on peut exécuter avec

mpirun -n « nombre de processeurs souhaité » ./run_speciesTransport

4 - Performances et résultats

Dans cette section, on va examiner les performances finales de notre programme.

Tous les temps d'exécution de notre programme mesurés dans les graphiques suivants ont été pris 10 fois et ont fait l'objet d'une moyenne. De plus, toutes les statistiques ont été réalisées en simulant une espèce, donc si on veut connaître le temps total pour simuler le logo ISC, il suffit de multiplier par 6.

On commence par une approche globale en regardant combien le temps total de notre système évolue au fur et à mesure que la taille du problème augmente.

Dans le premier graphique en bas, on va voir comment notre code parallèle est optimisé par rapport au code séquentiel de base. On aura trois courbes représentant respectivement le temps en séquentiel, le temps en parallèle avec un CPU et le temps avec 32 CPUs pour la partie mpi.

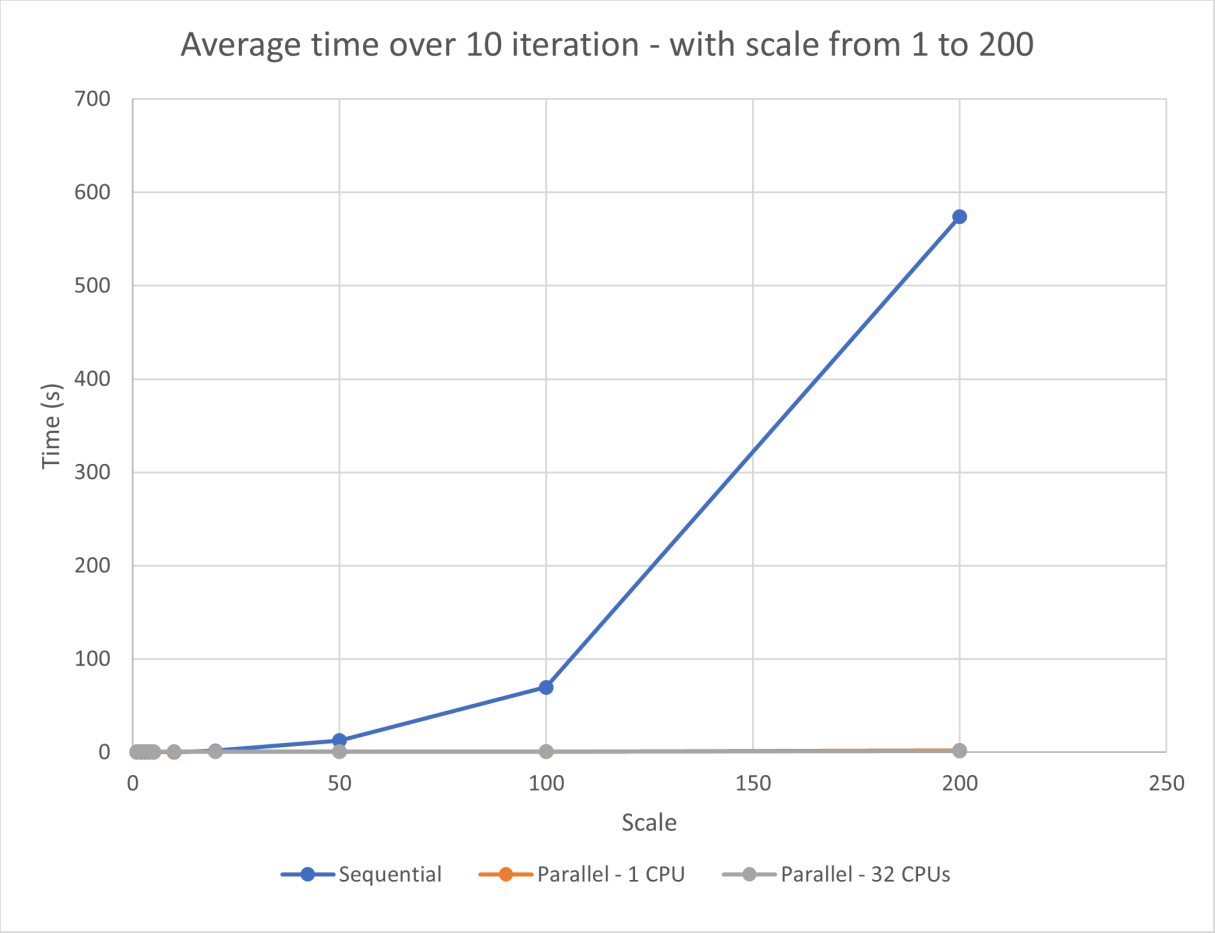


Figure 1 - Comparaison globale des performances du système avec un maillage de différentes tailles qui vont de 1 à 200

On peut constater que notre code parallèle devient très efficace par rapport au code séquentiel pour les grandes tailles. Mais comment se comportera-t-il pour des tailles plus petites ? Pour le comprendre, on analysera par la suite le même graphique, mais en ne retenant que les problèmes de plus petite taille.

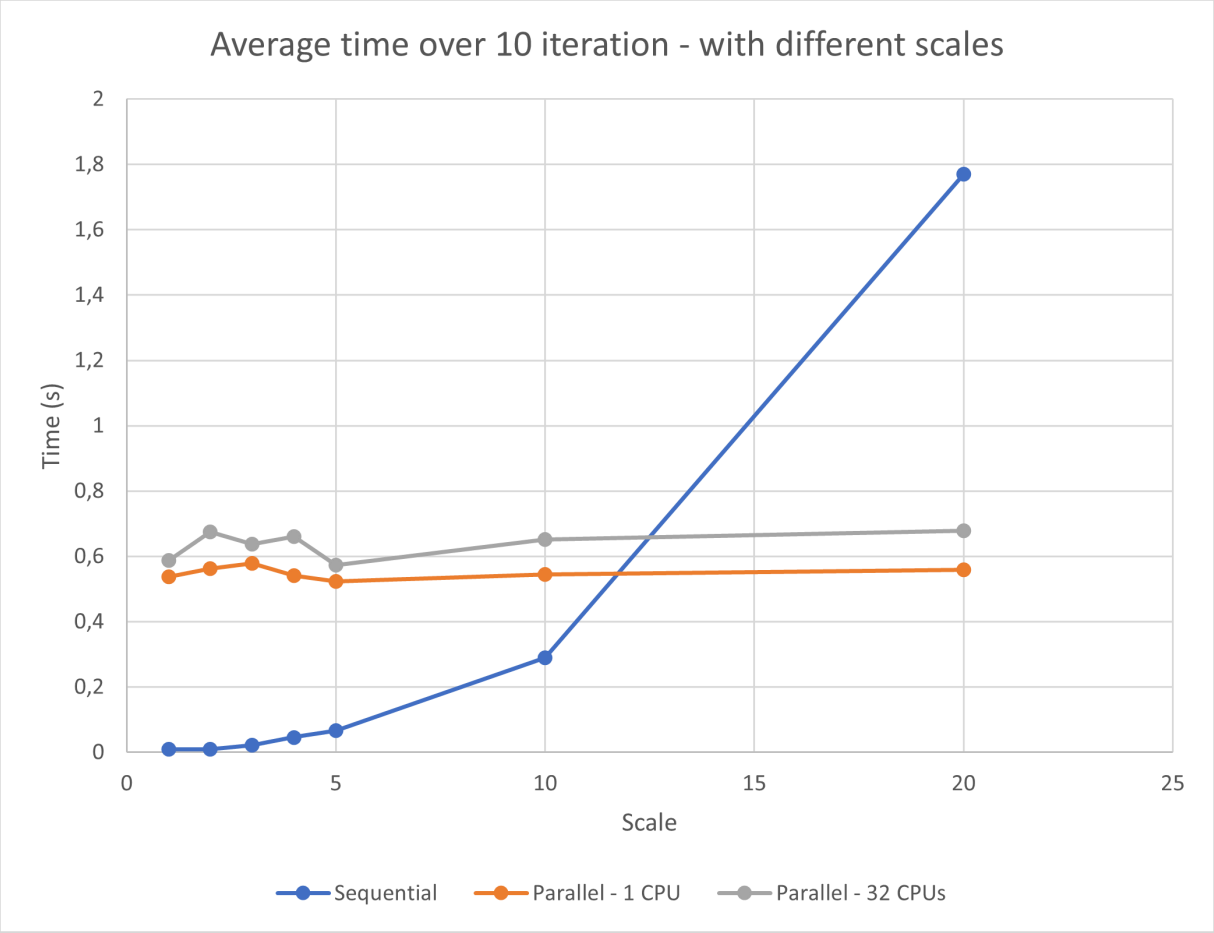


Figure 2 - Comparaison globale des performances du système avec un maillage qui varie avec des taille plus petites

Ce graphique montre une chose intéressante, c'est-à-dire que le temps séquentiel pour un problème de petite taille est bien meilleur que les deux temps calculés en parallèle. En fait, on verra que pour un problème de taille très baisse, le temps reste presque constant. Ceci est dû à la parallélisation avec CUDA. En fait, si on étudie le code avec un analyseur de performance GPU (dans mon cas nvprof) on peut voir que la partie qui prend le plus de temps à s'exécuter est la copie des résultats du gpu vers l'hôte. Malheureusement, cette opération sera toujours lente n'importe pas quelle est la taille du problème et on commence à voir un avantage de cette opération que pour des problèmes de taille plus grande, voir Figure 1. Une autre chose qu'on peut remarquer est que on a un temps de calcul légèrement meilleur avec seulement 1 CPU pour la partie de parallélisation mpi par rapport à 32 CPUs. Cela est peut être dû au fait que l'initialisation d'un si grand nombre de processus prend du temps, et si la taille du problème est petite, cela peut ne pas en valoir la peine.

Il serait maintenant intéressant de voir quand l'ajout de processeurs pour la partie mpi peut conduire à une amélioration de la performance du code. Pour cela, il est nécessaire d'analyser des tailles plus grandes du problème.

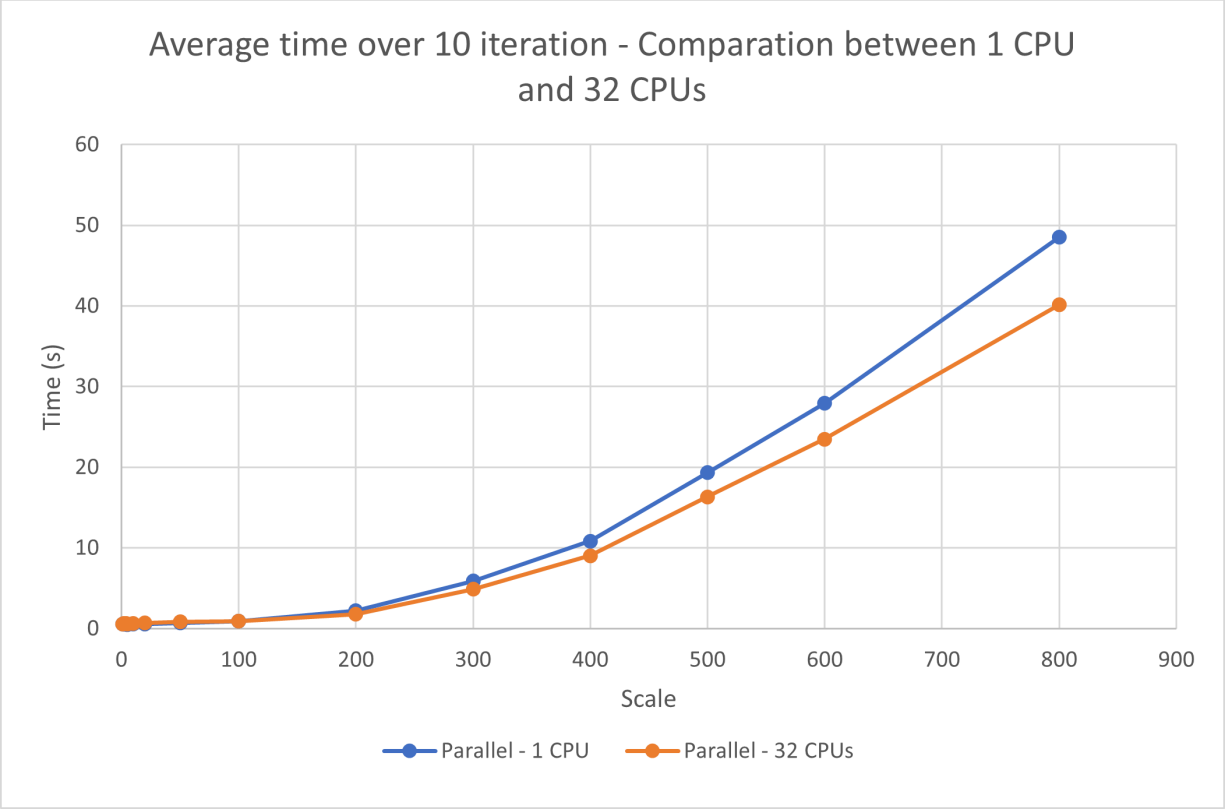


Figure 3 - Comparaison des résultats avec 1 et 32 CPUs

On peut voir que l'utilité d'utiliser plus de processeurs avec mpi se manifeste pour des problèmes de très grande taille, et on pourra considérer que le simple fait d'introduire CUDA dans notre code conduira à une bonne amélioration de nos performances. Par exemple, si on prends 200 comme taille du problème, en séquentiel on a un temps d'exécution de 573 secondes, parallélisé avec 1 CPU on a un temps de calcul de 2,21 secondes et avec 32 CPU de 1,79 secondes. On peut donc affirmer que dans ce cas notre programme en parallèle avec 1 CPU est environ 260 fois plus rapide que celui en séquentiel et que celui avec 32 CPU est environ 320 fois plus rapide.

4.1 - Performance de l'écriture des fichiers

On va maintenant étudier les performances de la partie parallèle pour l'écriture des fichiers avec mpi. Tout d'abord, il est intéressant de faire un strong scaling pour évaluer de combien notre programme peut devenir plus rapide en augmentant le nombre de cœurs. Le strong scaling consiste en fait à fixer la taille du problème et à augmenter le nombre de cœurs. Pour vérifier dans quelle mesure notre code peut bénéficier de l'augmentation du nombre de cœurs, on peut utiliser la mesure du speedup qui est égale à :

Speedup = T1/Tn

ou

T1 = temps en séquentiel dans ce cas a été utilisé MPI avec un processeur

Tn = temps avec n processeurs

Le speedup idéal serait égal au nombre de processeurs employés. Voyons maintenant comment cela se passe dans la réalité.

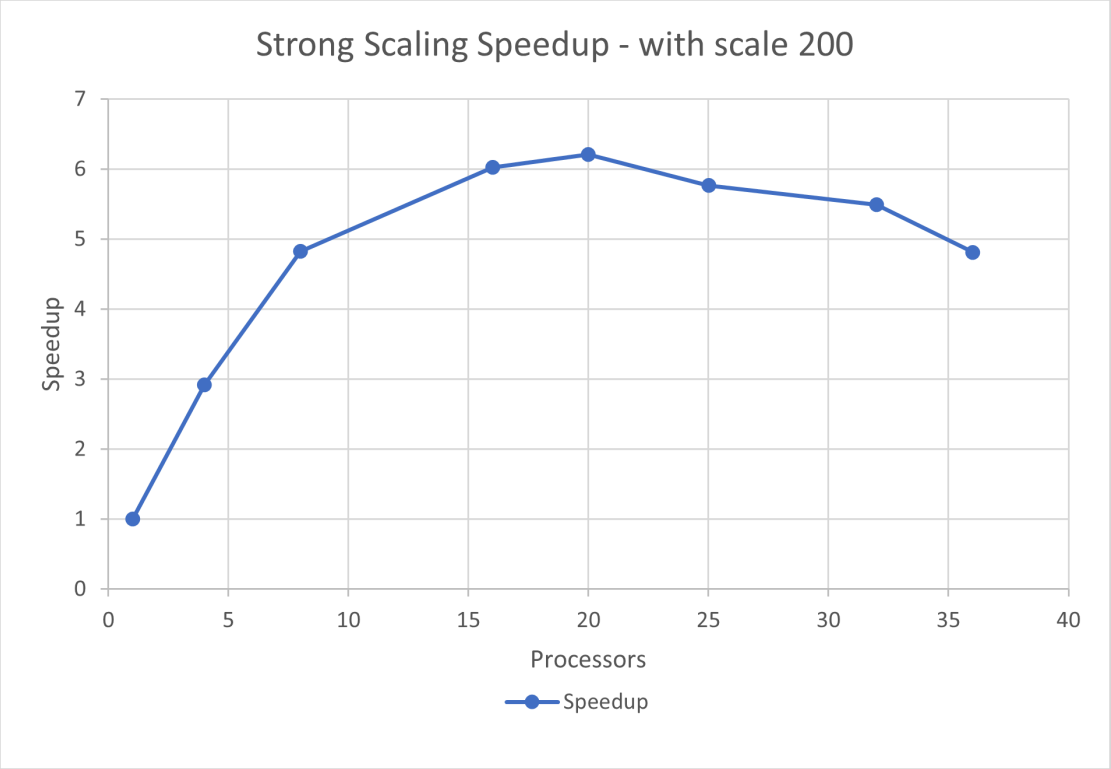


Figure 4 - Speedup strong scaling avec un maillage de 200x200

On voit que notre speedup réel reste loin de la valeur idéale, mais on voit quand même que dans tous les cas notre programme parallèle arrive à exploiter l'augmentation des cœurs (speedup supérieur à un) en s'exécutant mieux à 20 processeurs. C'est pourquoi dans les métriques ci-dessus, on a pas utilisé le nombre maximum de processeurs (36) pour mesurer le temps d'exécution total mais un nombre de processeurs un peu plus baisse, c'est-à-dire 32, qui est pas les plus optimal mais meilleur que le nombre maximum de processeurs.

On peut ensuite examiner l'efficacité qui est donnée par :

*Efficacité = n*T1/ Tn*

On peut mesurer à quel point le système accélère la résolution d'un problème taille fixe en augmentant le nombre de processeurs. Pour une parallélisation parfaite, elle devrait être égale à 1 même si on augmente le nombre de processeurs.

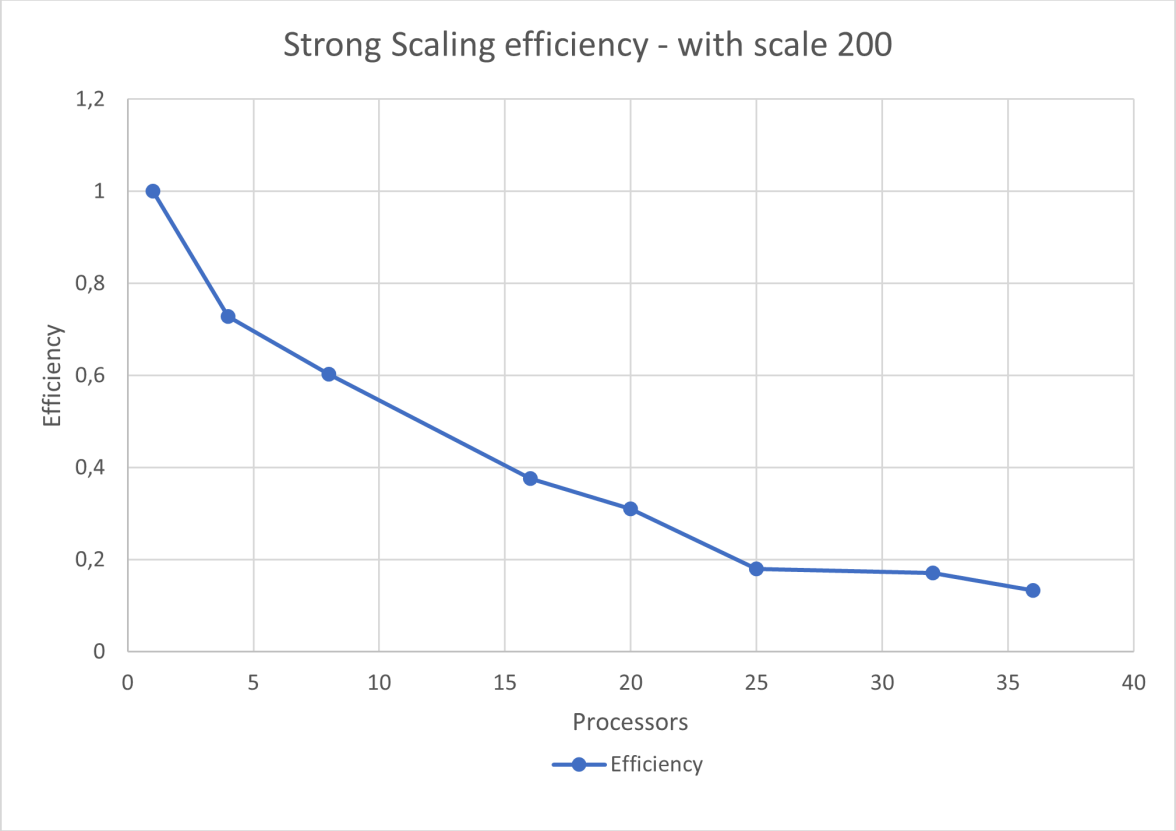


Figure 5 - Efficacité strong scaling avec un maillage 200x200

On constate une assez bonne efficacité, supérieure à 0,5 jusqu'à environ 12 processeurs. Elle se stabilise ensuite lorsque le nombre de processeurs est supérieur à 25, de sorte que l'ajout de processeurs n'entraînera pas d'amélioration des performances, mais seulement un gaspillage de ressources.

Enfin, on va aborder l'étude du problème par l'application du weak scaling. Cette approche consiste à augmenter la taille du problème proportionnellement au nombre de processeurs. Comme pour le strong scaling , on peut calculer l'efficacité que, dans ce cas, est donnée par:

Efficacité = T1/ Tn

Elle mesure la capacité du système à s'adapter à des problèmes plus grandes et à un plus grand nombre de processeurs.

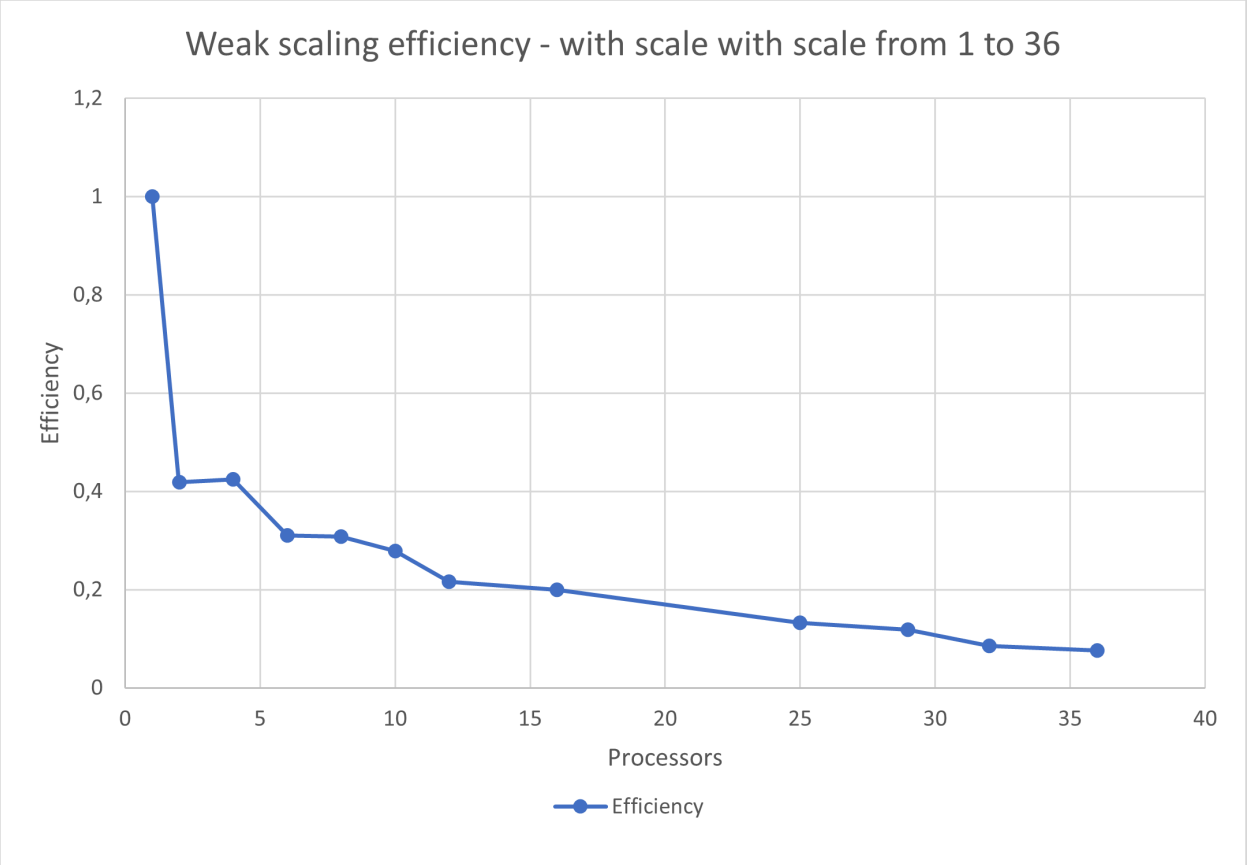


Figure 6 - Efficacité weak scaling avec un maillage qui varie de 1x1 à 36x36

Ce graphique indique que l'efficacité chute très rapidement, ce qui signifie que notre problème avec l'augmentation des processeurs proportionnelle à la taille du problème n'utilise pas très bien les ressources. On peut voir que notre efficacité est plus petit de 0,2 avec un nombre de processeurs plus grand de 10.

5 - Conclusion

En conclusion, on a un programme qui permet de simuler 6 espèces différentes et de les écrire dans des fichiers vtk. De plus, les espèces à simuler peuvent être facilement changées en modifiant le fichier d'initialisation. Cependant, le programme présente toujours un problème lorsqu'on veut l'exécuter sur plusieurs processeurs pour la partie écriture des fichiers. Ce problème est probablement dû à un mauvais calcul de l'offset pour l'écriture ou à une mauvaise répartition de l'offset entre les différents processus. Cela conduit à des valeurs incorrectes dans les petites parties des tableaux divisés et donc à une écriture incorrecte. Si l'on examine les performances, il faut voir que ce problème, même s'il est présent, n'est pas si grave car le code commence à bénéficier de manière significative du travail sur plusieurs processeurs pour des problèmes de taille très élevée. Par exemple, pour un maillage de 800x800,(voir Figure 3) on a 40 secondes en parallèle avec 32 CPU et 47,5 secondes avec 1 CPU. De plus, sur des tailles de problèmes plus petites (inférieures à 300), c'est même désavantageux. Donc, même si cela reste un problème, ce n'est pas très impactant en termes de performance.

Annexes

Table des figures

Table des listings