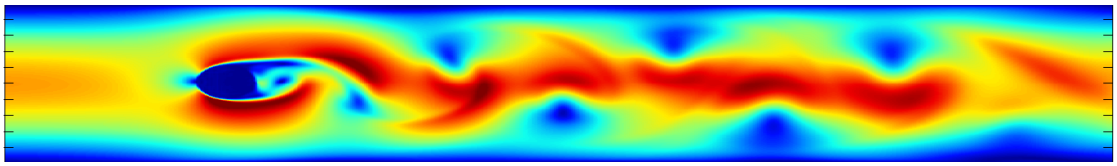
Université Paris-Saclay

Technique d'Optimisation de la Parallélisation

Optimisation de programme :

Tourbillons de Karman



Auteur :
LOZES Benjamin

1er mai 2022

université PARIS-SACLAY

Table des matières

0 Introduction

Le programme qui nous a été donné consiste à simuler le déplacement de fluides après impact avec un obstacle. Nous cherchons donc à observer la formation de tourbillons de Karman.

Ce genre de simulation nécessite un pas de temps très court et donc un nombre d'itérations très élevé. C'est donc pour cela qu'il a été décidé de permettre une éxecution multi-processus, afin de répartir la charge de travail de façon parallèle.

Ce programme sera implémenté et optimisé en C uniquement, bien qu'un partie du travail d'optimisation consistera à l'analyse du code assembleur généré.
Toutes les mesures ont été réalisées sur une machine Knights Landing (KNL) 6.2 avec les compilateurs GNU-C et OneAPI-ICC, et donc les implémentations MPICH et IntelMPI ; les options seront également précisées ; et tout threading sera réalisé avec la bibliothèque OpenMP.

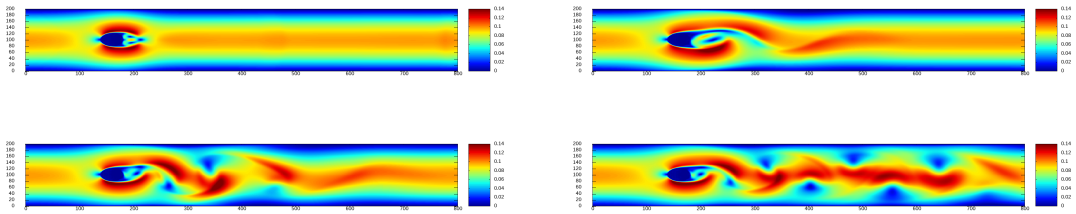


FIGURE 1 – Visualisation de la simulation

Toutes les données, graphes, mesures, etc. de ce rapport se baseront sur la configuration fixée (cf 6.1) sur un noeud Intel Kanon Lake (cf 6.2) et l'implémentation MPI de MPICH 3.3.2. À l'exception de la mesure avec 75 processus où nous utiliserons une maille de 1050 de large. De plus, chaque résultat de simulation a été vérifié par checksum et par comparaison visuelle afin de s'assurer de la cohérence de la simulation entre les différentes méthodes de simulation.

Une dépot Github est également disponible ici afin de consulter plus en détail le travail qui a été réalisé.

1 Distribution mémoire & Alignement

1.1 Répartition en grille

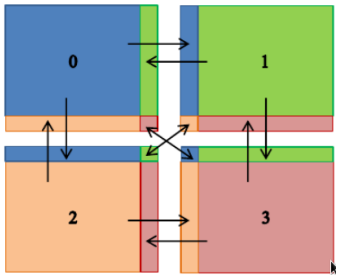


FIGURE 2 – Communication entre processus, répartis en grille, avec (gauche) et sans (droite) cellules fantômes

Le programme initial qui nous a été soumis implémentait un découpage de la maille en grille, cela permet une scalabilité quasi-infinie avec un découpage en sous mailles de même ratio. Néanmoins, cela pose un problème quant à l'accès mémoire des cellules, ne pouvant pas être alignées à la fois en colonne et en ligne.

En effet, les cellules étant allouées de façon contigues en mémoire, il est fondamentalement impossible (avec un format classique), de récupérer à la fois une suite de cellules d'une même colonne et une suite de cellules d'une même ligne.
Deux options s'offrent alors à nous :

- Communication multiple par cellules non-contigues
- Communication unique d'un nouveau vecteur de cellules recopiées au préalable, et à recopier après réception.

Un découpage 1D de la maille totale, soit en colonne, soit en ligne, serait donc une alternative logique pour améliorer les accès mémoire.

1.2 Répartition verticale

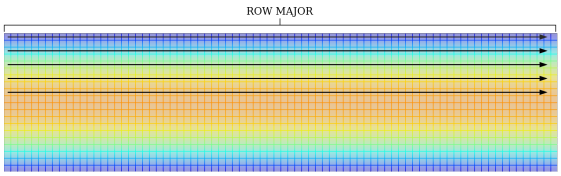


FIGURE 3 – Allocation ROW_MAJOR

Un découpage vertical permettrait de palier aux problèmes des caches vu précedemment. Néanmoins, la simulation étant prévue pour être plus large que haute, il serait illogique de découper dans ce sens.
À titre d'exemple, pour une configuration classique de 800x200, avec 200 processus, chaque processus n'aurait qu'une cellule de hauteur à traiter, réduisant grandement l'efficacité potentielle des caches et les possibilités de threading.

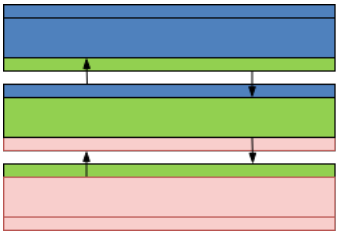


FIGURE 4 – Découpage vertical (allocation ROW_MAJOR)

1.3 Répartition horizontale

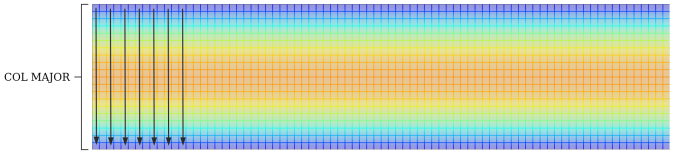


FIGURE 5 – Allocation COL_MAJOR

Un découpage horizontal, dans le cas de cette simulation, permettrait la meilleure utilisation possible des caches, du threading et notamment de la taille des message MPI à envoyer.

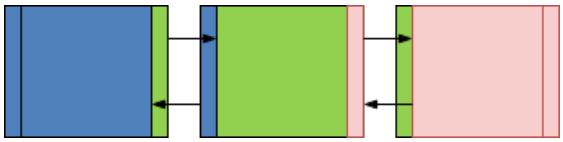


FIGURE 6 – Découpage horizontal (allocation COL_MAJOR)

Cela nous amène à notre section suivante, qui concerne le type de communication à utiliser pour envoyer ces bandes de cellules contigues.

2 Types de communication

2.1 Code de diffusion 2D

2.1.1 Point-à-point - sans buffer / alignement

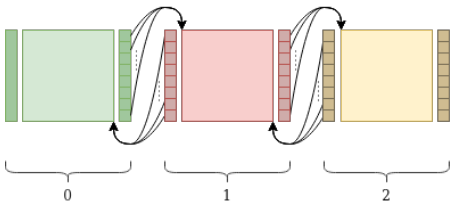


FIGURE 7 – Communications point-à-point, sans buffer

Communications réalisées dans le code initial, il s'agit d'envoyer une à une, potentiellement de manière asynchrone, chaque cellule. Comme étudié précédemment, cela n'est pas utile avec notre découpage.

Algorithm 1 Communication point-à-point d'une bande non-contigue

for $cell \in band$ **do**
  MPI_Sendrecv($cell$, $neigh_i$)
end for

2.1.2 Point-à-point - avec buffer / alignement

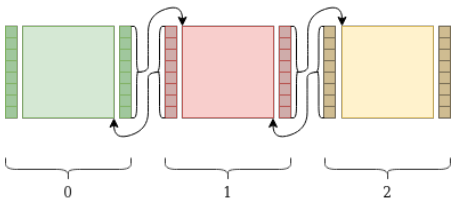


FIGURE 8 – Communications point-à-point, avec buffer

Première optimisation réalisées sur les communications, il s'agit d'envoyer les cellules par paquet/bandes. Comme étudié précédemment, cela justifie parfaitement notre découpage 1D.

Algorithm 2 Communication point-à-point point-à-point d'une bande contigue

MPI_Sendrecv($band$, $neigh_i$)

2.1.3 Collective - topologie graphe

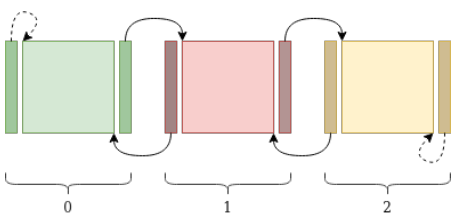


FIGURE 9 – Topologie graphe pondéré à 3 noeuds

Tentative d'optimisation avec des communications par *topologie de graphe distribué*, dépend fortement de l'implémentation MPI.
Implémentation ici.

Algorithm 3 Communication par graphe de bandes contigues

MPI_Dist_graph_create(*neighbors*)
...
MPI_Neighbor_Alltoall(*bands*)

2.1.4 Collective - topologie cartésienne

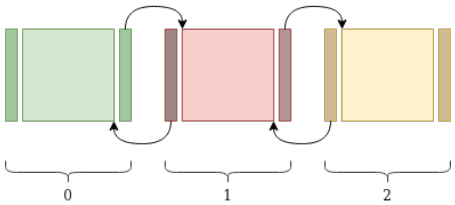


FIGURE 10 – Topologie cartésienne 1D à 3 noeuds

Tentative d'optimisation avec des communications par *topologie cartésienne à 1 dimension*, dépend fortement de l'implémentation MPI.
Implémentation ici.

Algorithm 4 Communication par topologie cartésienne 1D de bandes contigues

MPI_Cart_create(1)
...
MPI_Neighbor_Alltoall(*bands*)

2.1.5 Analyse des performances

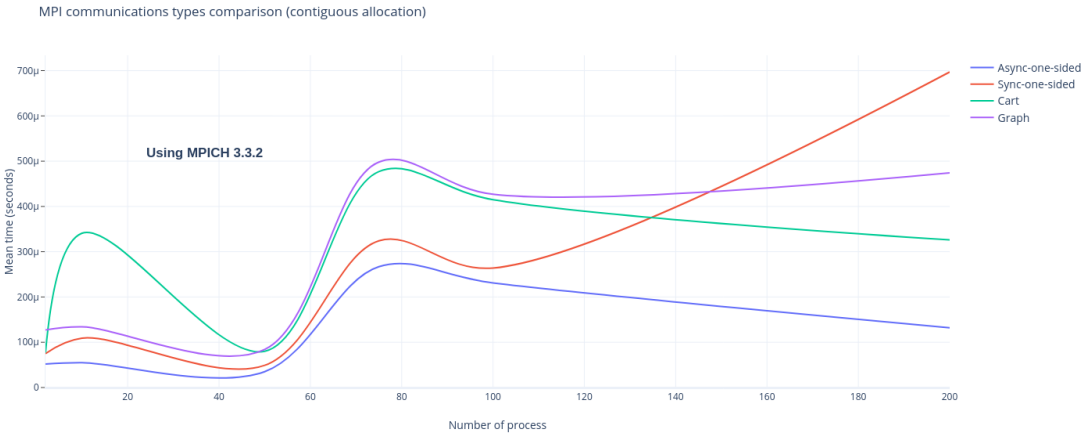


FIGURE 11 – Durée moyenne d'une communication (pour la configuration fixée)

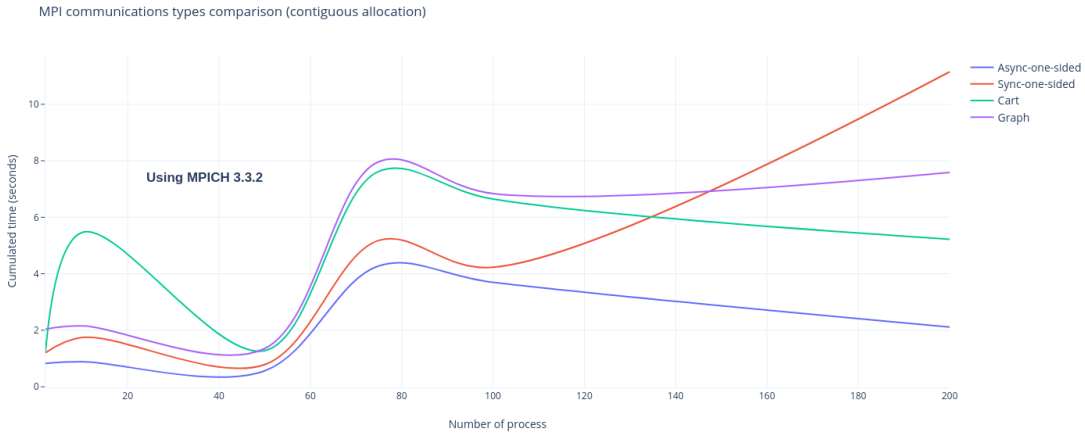


FIGURE 12 – Durée cumulées des communications (pour la configuration fixée)

On remarque alors clairement que les communications point-à-point unitaires sont les plus rapide que toutes les autres. Notamment, on constate la stabilité des mesures, la durée cumulée n'ayant pas été calculée par multiplication des moyennes bien entendu.

2.2 I/O

Les I/O sont une composante essentielle de cette simulation, ils sont certes moins fréquemment appelés mais bien plus long. Nous allons étudier l'optimisation possible que nous pourrions apporter à cette synchronisation.

Précision : la seule méthode d'I/O que j'ai pu/eu le temps d'implémenter est celle de base. Décrite dans le schéma suivant.

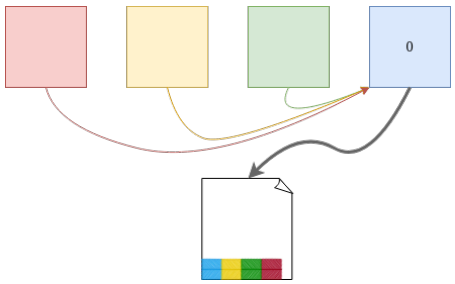


FIGURE 13 – Méthode d'I/O implémentée

2.2.1 Point-a-point

Comme décrit précédemment, il s'agit là d'envoyer la totalité de la maille locale avec un envoi point-à-point, vers le processus 0.

2.2.2 Collective - gather

Il existe également l'API *MPI_ Gather* permettant de faire la même communication collective de façon plus optimisée. C'est cette méthode que nous choisissions d'utiliser après comparaison des performances.

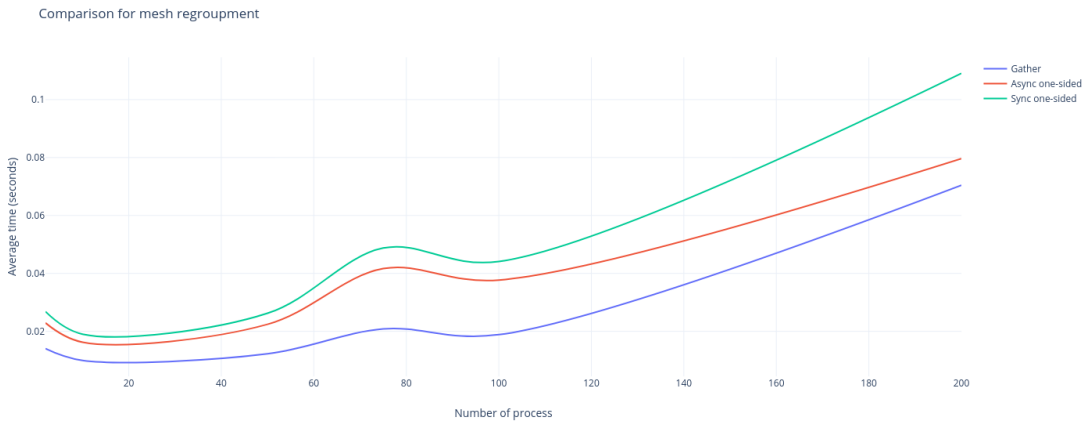


FIGURE 14 – Comparaison des performances pour la récupération des mailles dans le processus maître (0)

2.2.3 Collective spécifique aux fichiers

Cette section ne traitera que théoriquement de l'optimisation possible que représente les I/O spécifiques à MPI, la simulation en l'état ne permettant pas facilement de l'implémenter, et n'ayant découvert ces fonctionnalités que trop tard pour entreprendre une refactorisation complète.
J'ai néanmoins pu essayer l'API *MPI_ File_ write_ ordered* par curiosité, nous obtiendrions une amélioration de l'ordre de 23%, temps d'écriture disque comprise.

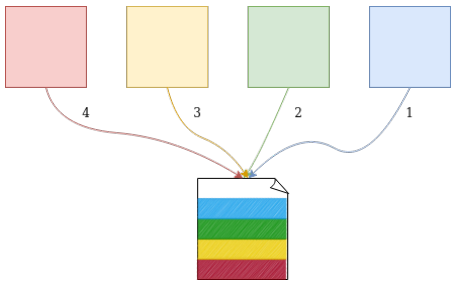


FIGURE 15 – MPI_File_write_ordered

2.2.4 I/O parallèle

Cette section ne traitera que théoriquement de l'optimisation possible que représente l'utilisation d'I/O parallèle. Le serveur utilisé n'étant pas équipé de système le permettant, il aurait inutile voir même risqué pour la stabilité serveur de l'implémenter.

Description graphique :

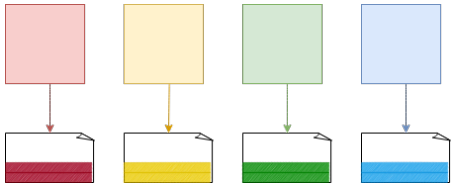


FIGURE 16 – Schématisation des I/O parallèle

De plus, voici quelques libraries optimisées pour I/O et I/O parallèles qui seraient intéressantes pour cette simulation :

— NetCDF
— HDF5
— IO-Sea

3 Optimisations diverses

3.1 Typage

Une vérification sur la cohérence des types a été effectuée, en effet, les 'cast implicite' sont souvent une source de ralentissement indésirée du programme.
Une erreur de ce genre est présente dns le programe, dans la routine *compute_ cell_ collision*, où l'on utilise des **double** au lieu de **int**, pour calculer des indices de boucle.

Résultat : durée de calcul de la routine réduite de 15%

Correctif ici

3.2 Déroulage de boucles

Utilisant des tailles fixes pour différentes entités de la simulation telles que les cellules ou la dimension, il est possible de dérouler manuellement les boucles liées à ces valeurs. À noter que la compilation se charge normalement de ce genre d'optimisation à notre place.

Exemple avec la routine compute_ bounce_ back ici

3.3 Système de session (ou Breakpoints)

Fonctionnalité non implémentée car non essentielle au vu du temps nécessaire pour effectuer une simulation dite "complète". Toutefois, ce système serait intéressant à facilement implémentable si la simulation venait à devenir interactive.

3.4 Implementation MPI

Toutes les mesures présentées dans ce rapport ont été effectuées avec MPICH 3.3.2. Ce choix est du à sa supériorité dans la totalité des tests menés sur cette simulation, OpenMPI ayant un ralentissement d'au moins 7%, et IntelMPI de 4% et ceux malgré le fait que la simulation ait lieu sur un noeud entièrement Intel.

3.5 Analyse de programme avec Maqao

```
#########################################################################################################################################################
#  Cluster  |  Walltime (s)  |  Exe (%)  |  MPI (%)  |  OMP (%)  |  System (%)  |  Pthread (%)  |  IO (%)  |  String (%)  |  Memory (%)  |  Others (%)  #
#########################################################################################################################################################
#  Cluster  |  92.41         |  93.44    |  4.88     |  0.36     |  0.20        |  0.18         |  0.07    |  0.34        |  0.10        |  0.43        #
#########################################################################################################################################################
```

FIGURE 17 – Analyse Maqao

4 Threads

4.1 Diffusion 2D parallèle

L'ajout de la librarie OpenMP pour paralléliser localement les différentes boucles du programme fut une étape importante dans l'optimisation du programme. Regardons tout 'abord, avec le graphique suivant les raisons qui nous poussent à utiliser les threads.

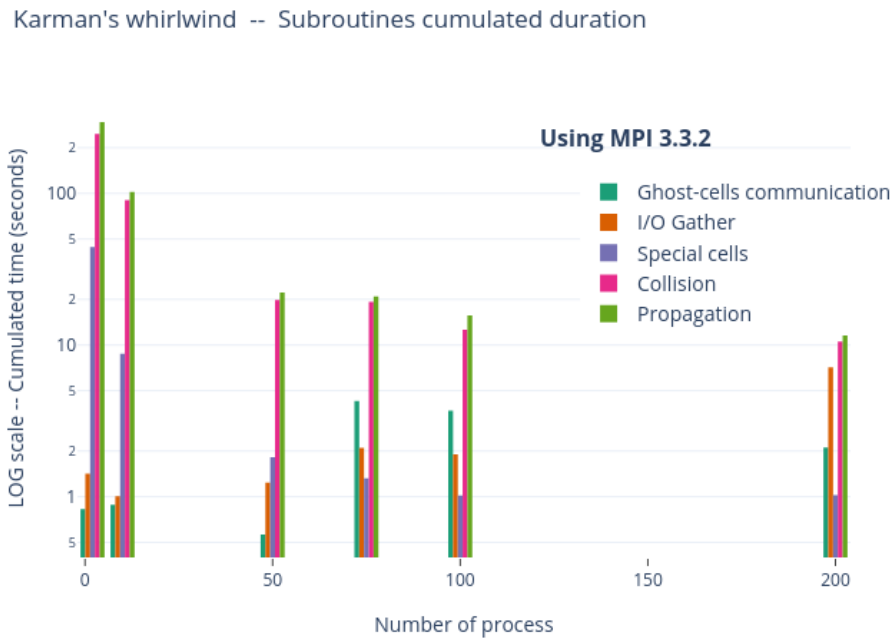


FIGURE 18 – Durée des principales routines de cette simulation (échelle log en Y)

Ce graphique nous apporte deux informations essentielles :

— 1. La durée des routines de simulation varie beaucoup selon le nombre de processus, et donc la taille de leurs boucles internes.

— 2. Nous passons plus de temps à calculer les résultats de la simulation qu'à les communiquer.

Le premier point est un indice pour le choix du threading, en effet, nous nous doutons que le threading de minuscule boucle, réduite par le nombre de processus important, n'aura que très peu d'effet positif, voir des effets négatifs à cause de l'*overhead* OpenMP.

Le second point nous indique que le travail d'optimisation réalisé précédemment à porté ses fruits, passer plus de temps à communiquer qu'à calculer, pour un code de simulation serait un signal alarmant quant à la qualité du code produit.

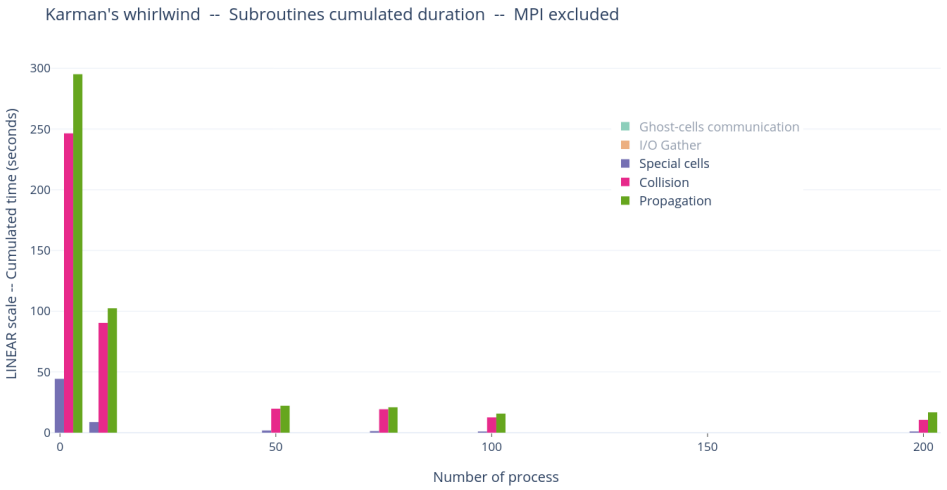


FIGURE 19 – Durée des principales routines de cette simulation, hors MPI (échelle linéaire en Y)

Résultats :
— Threading de *propagation* : 50.50% d'accélération, pour 75 processus.
— Threading de *collision* : 85.11% d'accélération, pour 75 processus.
— Threading de *special_ cells* : 40% de ralentissement, pour 75 processus.

4.2 Découpage en tâches

Cette section traîte du threading avec la librarie *pthread* de la phase d'I/O, en tant que tâche parallèle à la simulation.
Une tentative d'implémentation à vu le jour ici, mais elle fut infructueuse. La tentative n'a pas fonctionnée à cause d'un manque de connaissance sur l'hybridation MPI+pthread+OpenMP, mais n'étant pas primordiale pour améliorer les performances de l'ensemble du programme, j'ai préféré reporter cette idée.

Description schématique :

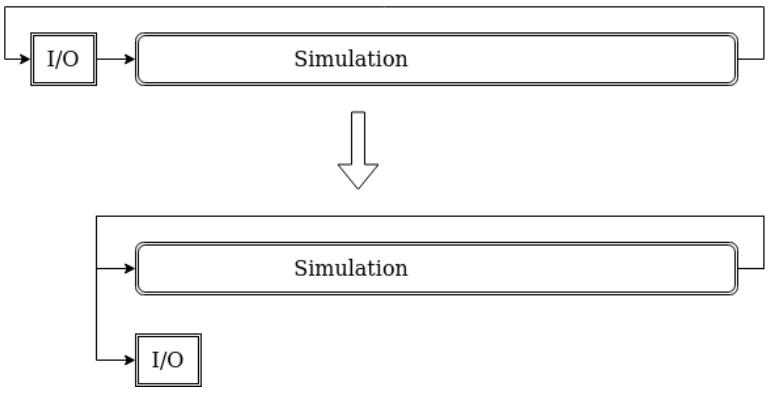


FIGURE 20 – Parallélisation complète de la phase d'I/O

5 Conclusion & Scalabilité

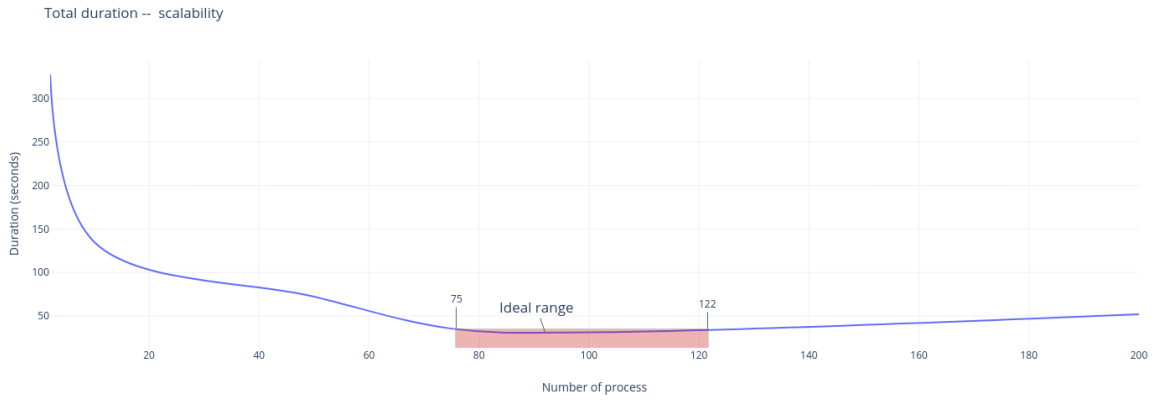


FIGURE 21 – Durée finale de la simulation optimisée + scalabilité

C'est avec un temps de simulation réduit d'un facteur 24 (difficile de situer la première version stable au vu du nombre de bugs initiaux), que s'achève cette optimisation.

Elle m'aura permis d'étudier les différentes approches d'utilisation de MPI, et par dessus tout de l'hybridation, rarement abordée jusque là dans nos cours. Je regrette de ne pas avoir pris plus de temps pour finir cette optimisation qui pourrait être bien meilleure encore, notamment en passant sur du calcul GPU ou encore avec des languages plus adaptés à ce genre de micro-codes tel que Julia. Les idées ne m'ont pas manquées, le temps malheureusement, si.

6 Spécifications

6.1 Configuration

```
iterations           = 16000
width                = 1000
height               = 500
reynolds             = 500
inflow_max_velocity  = 0.100000
output_filename      = resultat.raw
write_interval       = 160
```

FIGURE 22 – Configuration (pour toutes les mesures)

6.2 Cluster

— Serveur knl01 : Intel Knights Landing

Modèle : Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
Architecture : x86_64
Fréquence : 1400.00 MHz
Politique d'alimentation : "Performance"
Taille totale du cache : 1024 KB
Taille cache L1d : 2MiB ⟶ 2097 KB
Taille cache L2 : 32MiB ⟶ 33 554 KB
Attributs importants : sse, sse2, sse3, fma, sse4, avx, avx2, avx512
Alignement du cache : 64

6.3 Graphiques et données

Tous les graphiques utilisés nous ont été fournis ou ont été générés par ma personne, il en va de même pour les données de ces graphiques et schémas.