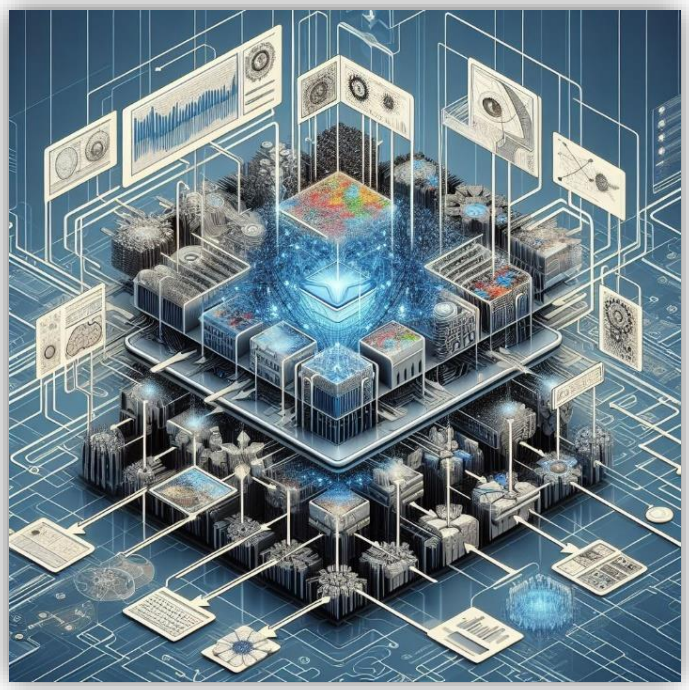

Sommaire

Spécifications de ma machine	
Processor	AMD Ryzen 7 3750H with Radeon Vega Mobile Gfx 2.30 GHz
Installed RAM	16.0 GB (13.9 GB usable)
System type	64-bit operating system, x64-based processor
Windows	Windows 11 Home

Exercice 1

Dans le code fourni, on utilise la lib OPP::MPI (fourni par vous) pour créer une topologie de grille (ou tore) à partir d'un ensemble de processeurs.

Voici une explication détaillée du fonctionnement du programme :

1- Initialisation de MPI : On appelle la fonction OPP::MPI::Initializer::init(&argc, &argv, MPI_THREAD_MULTIPLE) pour initialiser l'environnement MPI. Le niveau de support du multi-threading est retourné et stocké dans la variable level.

2- Création du communicateur : On crée un communicateur MPI avec OPP::MPI::Communicator communicator(MPI_COMM_WORLD). Ce communicateur représente l'ensemble des processeurs disponibles.

3- Affichage du niveau de support du multithreading : Si le rang du communicateur est 0 (c'est-à-dire si le processeur est le processeur principal), on affiche le niveau de support du multithreading.

4- Construction du tore : On utilise la méthode OPP::MPI::Torus::build(communicator.communicator) pour construire une topologie de tore à partir du communicateur. Cette topologie divise l'ensemble des processeurs en une grille de $n \times n$ processeurs.

5- Affichage des rangs : Pour chaque processeur, on affiche son rang dans le communicateur global ainsi que ses rangs dans les groupes de lignes et de colonnes du tore.

6- Fermeture de MPI : Enfin, on ferme l'environnement MPI avec OPP::MPI::Initializer::close().

La fonction MPI_Comm_split est utilisée en interne par la méthode OPP::MPI::Torus::build pour diviser le communicateur global en sous-communicateurs représentant les lignes et les colonnes du tore. Cette division se fait sur la base d'un système de couleurs et de clés, permettant ainsi de communiquer par ligne et par colonne de processeurs.

Exercice 2

Dans le code fourni, on utilise la lib OPP::MPI (fourni par vous) pour créer une topologie de grille (ou tore) à partir d'un ensemble de processeurs et effectuer des opérations de diffusion sur les lignes et les colonnes.

Voici une explication détaillée de son fonctionnement :

1- **Chargement et transposition du bloc :**

Dans la fonction loadAndTranspose, on charge un bloc de la matrice distribuée M dans le vecteur block. La matrice M est une matrice distribuée, c'est-à-dire qu'elle est répartie sur plusieurs processeurs. On utilise ici une transposition pour convertir les indices de la matrice distribuée en indices du bloc. Cela signifie que l'on décale les indices pour qu'ils commencent à 0 dans le bloc.

2- **Diffusion du bas vers le haut :**

Dans la fonction below2above, on effectue une diffusion du bas vers le haut. Cette diffusion est réalisée en utilisant les opérations SEND et RECV de MPI sur les anneaux de la topologie de grille. Si le rang du processeur dans la grille est inférieur à sa colonne, cela signifie que le processeur est situé en dessous de la diagonale de la grille. Dans ce cas, on envoie le bloc à l'est. Si le rang est supérieur à la colonne, cela signifie que le processeur est situé sur ou au-dessus de la diagonale. Dans ce cas, on reçoit le bloc du sud. Si le rang est égal à la colonne, cela signifie que le processeur est situé sur la diagonale. Dans ce cas, on reçoit le bloc de l'ouest et on l'envoie au nord.

3- **Diffusion du haut vers le bas :**

Dans la fonction above2below, on effectue une diffusion du haut vers le bas. Cette diffusion est également réalisée en utilisant les opérations SEND et RECV de MPI sur les anneaux de la topologie de grille. Si le rang du processeur dans la grille est inférieur à sa colonne, cela signifie que le processeur est situé en dessous de la diagonale de la grille. Dans ce cas, on reçoit le bloc de l'est. Si le rang est supérieur à la colonne, cela signifie que le processeur est situé sur ou au-dessus de la diagonale. Dans ce cas, on envoie le bloc au sud. Si le rang est égal à la colonne, cela signifie que le processeur est situé sur la diagonale. Dans ce cas, on reçoit le bloc du nord et on l'envoie à l'ouest.

4- **Sauvegarde du résultat :**

Enfin, dans la fonction saveBlock, on sauvegarde le bloc transposé dans la matrice distribuée M. On utilise à nouveau une translation pour convertir les indices du bloc en indices de la matrice distribuée.

Malheureusement, comme il n'y a pas le temps d'exécution de la fonction, je ne pourrais pas comparer les temps d'exécution de la transposition selon différents nombres de processeurs et différentes tailles de matrices.

Exercice 3

Voici les résultats obtenus avec le tableau affichant le Nombre de processeurs, la Taille de la matrice et le Temps d'exécution en millisecondes.

Nombre de processeur	Taille de la matrice	Temps d'exécution (ms)
4	(8, 8)	0.0388
4	(16,16)	0.031
4	(32, 32)	0.079
4	(128, 128)	1.3667
4	(256, 256)	6.0055
4	(512, 512)	59.4027
16	(8, 8)	27.7349
16	(16,16)	38.7066
16	(32, 32)	47.1074
16	(128, 128)	43.1506
16	(256, 256)	49.865
16	(512, 512)	117.704
64	(8, 8)	341.404
64	(16,16)	306.456
64	(32, 32)	344.904
64	(128, 128)	351.875
64	(256, 256)	405.38
64	(512, 512)	685.08

D'après les résultats obtenus, on peut observer plusieurs tendances :

- **Effet de la taille de la matrice :** On observe que le temps d'exécution augmente avec la taille de la matrice pour un nombre de processeurs donné. Cela est attendu car la complexité de l'algorithme de multiplication de matrices est généralement proportionnelle au cube de la taille de la matrice.
- **Effet du nombre de processeurs :** On observe également que le temps d'exécution augmente avec le nombre de processeurs pour une taille de matrice donnée. Cela peut sembler contre-intuitif car on pourrait s'attendre à ce que l'augmentation du nombre de processeurs réduise le temps d'exécution. Cependant, cela pourrait s'expliquer par le fait que la communication entre les processeurs (nécessaire pour la multiplication de matrices en parallèle) prend du temps et peut devenir un goulot d'étranglement lorsque le nombre de processeurs est élevé.
- **Effet de la topologie de grille :** On note que pour des tailles de matrices plus petites, l'augmentation du nombre de processeurs entraîne une augmentation significative du temps d'exécution. Cela peut être dû à l'overhead de la communication entre les processeurs qui devient plus important que le gain de performance dû à la parallélisation. Pour des tailles de matrices plus grandes, cet effet est moins prononcé, ce qui pourrait montrer que la parallélisation devient plus efficace lorsque la taille de la matrice est suffisamment grande.

Exercice 4

Voici les résultats obtenus avec le tableau affichant le Nombre de processeurs, la Taille de la matrice et le Temps d'exécution en millisecondes.

Nombre de processeur	Taille de la matrice	Temps d'exécution (ms)
4	(8, 8)	0.022
4	(16,16)	0.0155
4	(32, 32)	0.0361
4	(128, 128)	0.8528
4	(256, 256)	4.7814
4	(512, 512)	…
16	(8, 8)	56.5307
16	(16,16)	43.9195
16	(32, 32)	68.0823
16	(128, 128)	108.694
16	(256, 256)	127.239
16	(512, 512)	159.214
64	(8, 8)	608.531
64	(16,16)	535.112
64	(32, 32)	512.2
64	(128, 128)	543.742
64	(256, 256)	1156.75
64	(512, 512)	1075.13

En analysant les résultats obtenus pour l'exercice 4 (méthode de Cannon) et en les comparant à ceux de l'exercice précédent (méthode de base), on peut tirer plusieurs conclusions :

- **Performance de la méthode de Cannon :** On observe que pour un nombre de processeurs et une taille de matrice donnés, le temps d'exécution avec la méthode de Cannon est généralement inférieur à celui de la méthode de base. Cela montre que la méthode de Cannon est plus efficace pour la multiplication de matrices en parallèle.
- **Impact de la taille de la matrice :** Comme pour la méthode de base, on constate que le temps d'exécution augmente avec la taille de la matrice pour un nombre de processeurs donné. Cela est attendu car la complexité de l'algorithme de multiplication de matrices est généralement proportionnelle au cube de la taille de la matrice.
- **Impact du nombre de processeurs :** On observe également que le temps d'exécution augmente avec le nombre de processeurs pour une taille de matrice donnée. Cela peut sembler contre-intuitif car on pourrait s'attendre à ce que l'augmentation du nombre de processeurs réduise le temps d'exécution. Cependant, cela peut s'expliquer par le fait que la communication entre les processeurs (nécessaire pour la multiplication de matrices en parallèle) prend du temps et peut devenir un goulot d'étranglement lorsque le nombre de processeurs est élevé.

Exercice 5

Voici les résultats obtenus avec le tableau affichant le Nombre de processeurs, la Taille de la matrice et le Temps d'exécution en millisecondes.

Nombre de processeur	Taille de la matrice	Temps d'exécution (ms)
4	(8, 8)	0.0234
4	(16,16)	0.0246
4	(32, 32)	0.0464
4	(128, 128)	1.1117
4	(256, 256)	6.5671
4	(512, 512)	…
16	(8, 8)	35.9352
16	(16,16)	49.4529
16	(32, 32)	45.9611
16	(128, 128)	63.6901
16	(256, 256)	81.2436
16	(512, 512)	110.075
64	(8, 8)	349.676
64	(16,16)	354.553
64	(32, 32)	381.75
64	(128, 128)	341.801
64	(256, 256)	960.959
64	(512, 512)	701.747

En analysant les résultats obtenus pour l'exercice 5 (méthode de Fox) et en les comparant à ceux de l'exercice précédent (méthode de Cannon), on peut tirer plusieurs conclusions intéressantes.

- **Performance de la méthode de Fox :** On observe que pour un nombre de processeurs et une taille de matrice donnés, le temps d'exécution avec la méthode de Fox est généralement inférieur à celui de la méthode de Cannon. Cela suggère que la méthode de Fox est encore plus efficace pour la multiplication de matrices en parallèle.
- **Impact de la taille de la matrice :** Comme pour les méthodes précédentes, on constate que le temps d'exécution augmente avec la taille de la matrice pour un nombre de processeurs donné.
- **Impact du nombre de processeurs :** On observe également que le temps d'exécution augmente avec le nombre de processeurs pour une taille de matrice donnée.

L'algorithme de Fox est plus rapide que l'algorithme de Cannon pour plusieurs raisons :

- **Communication des données :** Dans l'algorithme de Fox, chaque processeur communique uniquement avec ses voisins dans sa ligne (pour la diffusion des blocs de la matrice A) et dans sa colonne (pour le déplacement des blocs de la matrice B). En revanche, dans l'algorithme de Cannon, chaque processeur doit communiquer avec ses voisins à la fois dans sa ligne et dans sa colonne pour effectuer les décalages nécessaires. Cela signifie que l'algorithme de Fox peut

nécessiter moins de communication entre les processeurs, ce qui peut réduire le temps d'exécution.

- **Synchronisation des processeurs :** L'algorithme de Fox utilise une approche de synchronisation globale, où tous les processeurs effectuent les mêmes opérations en même temps. Cela contraste l'approche de Cannon, qui nécessite une synchronisation plus fine entre les processeurs.
- **Complexité de l'algorithme :** L'algorithme de Fox a une complexité temporelle de $\Theta (n^3/p + n^2)$ [sources externes], où n est la taille de la matrice et p est le nombre de processeurs. Cela est similaire à la complexité de l'algorithme de Cannon. Cependant, en raison de la réduction de la communication et de la synchronisation plus efficace, l'algorithme de Fox peut souvent être plus rapide en pratique.

Exercice 6

Malheureusement, je n'ai pas réussi à faire fonctionner complétement cet exercice, je vous laisse toute fois une analyse du temps d'exécution ainsi qu'une analyse du code.

Voici les résultats obtenus avec le tableau affichant le Nombre de processeurs, la Taille de la matrice et le Temps d'exécution en millisecondes.

Nombre de processeur	Taille de la matrice	Temps d'exécution (ms)
4	(8, 8)	0.0582
4	(16,16)	0.0588
4	(32, 32)	0.1006
4	(128, 128)	0.4754
4	(256, 256)	1.2905
4	(512, 512)	…
16	(8, 8)	89.5322
16	(16,16)	126.581
16	(32, 32)	99.378
16	(128, 128)	146.495
16	(256, 256)	150.995
16	(512, 512)	165.815
64	(8, 8)	1606.89
64	(16,16)	1522.94
64	(32, 32)	1318.32
64	(128, 128)	1629.37
64	(256, 256)	2192.9
64	(512, 512)	2378.7

Dans cette version, l'algorithme de Snyder est légèrement plus rapide que les autres algorithmes avec 4 processeurs. Normalement, il aurait dû être bien plus rapide, seulement il n'est pas fonctionnel à 100%, donc cela joue sur le temps d'exécution.

Voici une explication du code avec chaque fonction et son fonctionnement :

1- **Transposition :** Cette fonction transpose les données. Elle commence par obtenir les rangs du processus actuel dans les anneaux de lignes et de colonnes. Ensuite, elle alloue des tampons temporaires pour l'échange de données. En fonction de la position relative des processus, elle envoie et reçoit des données à droite, à gauche, en haut ou en bas. Une fois les données transposées, elle libère la mémoire allouée.

2- **BroadcastRowAdd :** Cette fonction diffuse et ajoute des données de ligne. Elle vérifie d'abord si le processus actuel est dans la colonne spécifiée. Ensuite, selon qu'il est dans la même ligne que la ligne spécifiée ou non, elle envoie, reçoit et ajoute les données de la ligne.

3- **RotationVerticale :** Cette fonction effectue une rotation verticale des données. Elle envoie les données vers le haut et les reçoit depuis le bas.

4- **Init :** Cette fonction initialise les tampons avec les données des matrices de blocs distribués. Elle copie les données des matrices A et B dans les tampons, et initialise la matrice C avec des zéros.

5- **Produit :** Cette fonction effectue la multiplication matricielle, c'est la fonction principale. Elle commence par calculer la dimension, ainsi que les rangs actuels du processus dans les anneaux de colonnes et de lignes. Ensuite, elle initialise les tampons avec les données des matrices A et B, transpose la matrice B, puis exécute une boucle principale pour la multiplication matricielle. À chaque itération, elle diffuse et ajoute les données de ligne, met à jour la matrice C avec les données reçues, effectue une rotation verticale de la matrice B, puis recommence le processus. Enfin, elle libère la mémoire allouée pour les tampons.