Travaux dirigés n°1

Xavier JUVIGNY

February 10, 2025

Contents

1 Produit matrice–matrice

Soient A et B deux matrices définies à l'aide de deux couples de vecteurs $\{u_A, v_A\}$ et $\{u_B, v_B\}$:

$$\left\{ \begin{array}{rcl} A & = & u_A.v_A^T \text{ soit } A_{ij} = u_{A_i}.v_{A_j} \\ B & = & u_B.v_B^T \text{ soit } B_{ij} = u_{B_i}.v_{B_j} \end{array} \right. .$$

On calcule le produit matrice–vecteur C=A.B à l'aide d'un produit matrice–matrice plein (complexité de $2.n^3$ opérations arithmétiques) et on valide le résultat obtenu à l'aide de l'expression sous forme de produit tensoriel de A et B :

$$\begin{array}{rcl@{\quad}rcl@{\quad}rcl} C & = & A.B & = & \left(u_A.v_A^T\right).\left(u_B.v_B^T\right) & = & u_A\left(v_A^T.u_B\right)v_B^T \\ & = & u_A\left(v_A|u_B\right)v_B^T & = & \left(v_A|u_B\right)u_A.v_B^T & & \end{array} .$$

Soit :

$$C_{ij} = \left(v_A|u_B\right)u_{A_i}.v_{B_j},$$

ce qui nécessite en tout $2.n + 2.n^2$ opérations arithmétiques (dont $2.n$ opérations pour le produit scalaire).

On se propose, par étape, de paralléliser en mémoire partagée le produit matrice–matrice fourni dans le fichier `ProdMatMat.cpp`. L'exécutable pour tester le produit matrice-matrice est `TestProduitMatrix.exe` :

1. Mesurez le temps de calcul du produit matrice–matrice donné en donnant en entrée diverses dimensions. Essayez en particulier de prendre pour dimension 1023, 1024 et 1025 (il suffit de passer la dimension en argument à l'exécution. Par exemple `./TestProductMatrix.exe 1023` testera le produit matrice-matrice pour des matrices de dimension 1023). En vous servant des transparents du cours, expliquez clairement les temps obtenus.

   Réponse : Pour $n = 1024$, le temps de calcul du produit de matrices est beaucoup plus long que pour $n = 1023$ et $n = 1025$. Cela se produit systématiquement pour des valeurs correspondant à des puissances de 2 supérieures à la taille de bloc (**block size**) du cache du processeur. Dans le cas de mon processeur, cette taille est de 64 octets.

   Ce problème s'explique par un alignement mémoire qui entraîne des conflits de cache constants. Comme 1024 est une puissance de 2, les adresses mémoire des éléments des matrices sont parfaitement alignées, ce qui peut rendre les accès moins efficaces en raison du **conflit de cache**. En revanche, pour $n = 1023$ et $n = 1025$, les accès ne suivent pas un schéma aussi rigoureusement aligné, ce qui réduit ces problèmes et permet d'obtenir de meilleures performances.

2. **Première optimisation :** Permutez les boucles en i, j et k jusqu'à obtenir un temps optimum pour le calcul du produit matrice–matrice (et après vous être persuadé que cela ne changera rien au résulat du calcul). Expliquez pourquoi la permutation des boucles optimale que vous avez trouvée est bien la façon optimale d'ordonner les boucles en vous servant toujours du support de cours.

   Réponse : L'ordre j, k, i est le plus efficace, car il fixe d'abord une colonne de B, puis une colonne de A, et enfin parcourt les lignes de cette colonne de A. Cette approche est optimale en raison de la façon dont les matrices sont stockées en mémoire sous la forme d'un vecteur unidimensionnel (`m_arr_coefs[row + col * nRows]`). Comme les éléments d'une même colonne sont contigus en mémoire, cet ordre de boucle permet de les accéder de manière séquentielle, exploitant ainsi la localité spatiale et réduisant les rechargements inutiles du cache. De plus, la matrice B est également accédée de manière contiguë, minimisant ainsi les erreurs de cache (**cache misses**). De cette façon, nous évitons les rechargements inutiles du cache et assurons un accès plus efficace aux données, améliorant ainsi de manière significative les performances du produit matriciel.

3. **Première parallélisation :** A l'aide d'OpenMP, parallélisez le produit matrice–matrice. Mesurez le temps obtenu en variant le nombre de threads à l'aide de la variable d'environnement `OMP_NUM_THREADS`. Calculez l'accélération et le résultat obtenu en fonction du nombre de threads, commentez et expliquez clairement ces résultats.

   Réponse : L'ajout de la directive # pragma omp parallel for avant la boucle externe est suffisant pour une parallélisation initiale, car cela distribue les itérations de la boucle entre les threads, permettant à différentes parties de la multiplication de matrices d'être traitées simultanément. Cela se traduit par une réduction du temps d'exécution et une utilisation plus efficace des ressources informatiques disponibles.

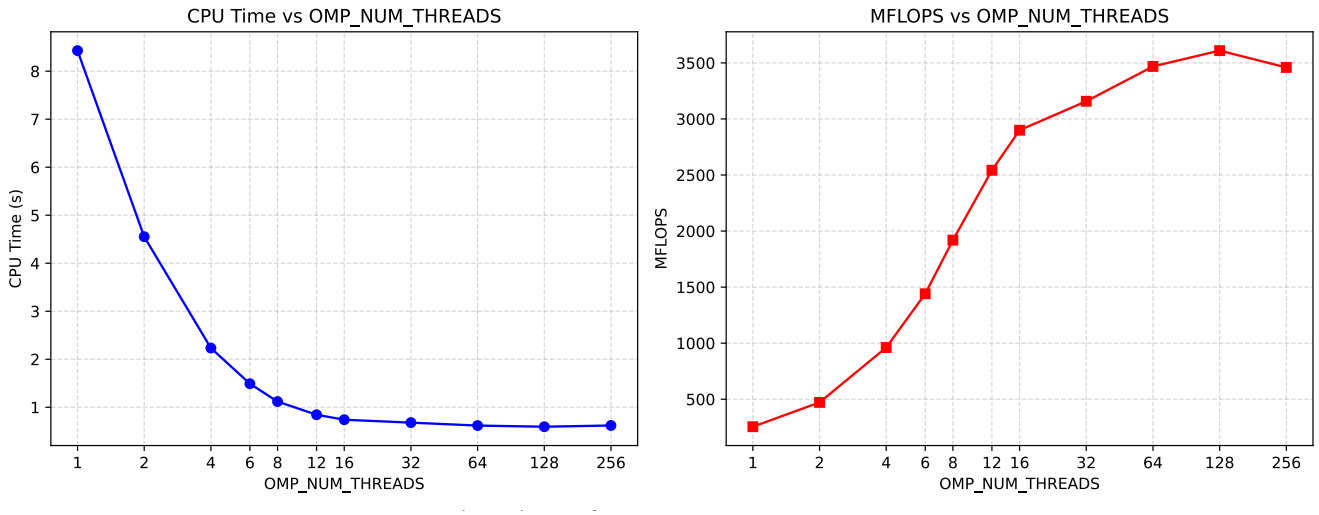


Figure 1: Speedup of matrix multiplication using OpenMP with varying thread counts.

4. Argumentez et donnez clairement la raison pour laquelle il est sûrement possible d'améliorer le résultat que vous avez obtenu.

   Réponse :

   Il est nécessaire d'utiliser correctement la fonction **prodSubBlock** en parallélisant le calcul de la matrice dans des régions indépendantes. Cela permet d'optimiser l'utilisation du cache entre les processeurs, car chaque processeur travaille sur une partie spécifique de la mémoire, évitant ainsi de charger les mêmes données dans plusieurs caches. Par conséquent, il n'y a pas besoin de transférer les mêmes informations entre la mémoire principale et les caches à plusieurs reprises, ce qui diminue la latence d'accès à la mémoire et améliore considérablement les performances grâce à une meilleure gestion des ressources et un meilleur partage du travail.

5. **Deuxième optimisation :** Pour pouvoir exploiter au mieux la mémoire cache, on se propose de transformer notre produit matrice–matrice "scalaire" en produit matrice–matrice par bloc (on se

servira pour le produit "bloc–bloc" de la meilleure version **séquentielle** du produit matrice–matrice obtenu précédemment).

L'idée est de décomposer les matrices A, B et C en sous-blocs matriciels :

$$A = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{N1} & & & A_{NN} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1N} \\ B_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ B_{N1} & & & B_{NN} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ C_{N1} & & & C_{NN} \end{pmatrix},$$

où A_{IJ}, B_{IJ} et C_{IJ} sont des sous–blocs possédant une taille fixée (par le programmeur).

Le produit matrice–matrice se fait alors par bloc. Pour calculer le bloc C_{IJ}, on calcule :

$$C_{IJ} = \sum_{K=1}^{N} A_{IK}.B_{KJ}.$$

Mettre en œuvre ce produit matrice–matrice en séquentiel puis faire varier la taille des blocs jusqu'à obtenir un optimum.

Réponse : Pour la prochaine étape de parallélisation, nous allons utiliser le code suivant :

Listing 1: Parallélisation avec OpenMP

```
for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock)
  for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock)
    for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock)
      prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
```

6. Comparer le temps pris par rapport au produit matrice–matrice "scalaire". Comment interprétez vous le résultat obtenu ?

   Réponse : Les tests ont montré qu'en augmentant progressivement la taille des blocs lors de la multiplication matricielle, on améliore significativement les performances par rapport à la méthode scalaire, grâce à une meilleure exploitation du cache mémoire. En effet, des tailles de blocs faibles entraînent des accès discontinus aux données, alors qu'une taille de 64 optimise la réutilisation des informations en cache, réduisant ainsi le temps d'exécution. Au-delà de cette valeur, les gains se stabilisent, indiquant que la taille de bloc optimale pour cette processeur se situe autour de 64.

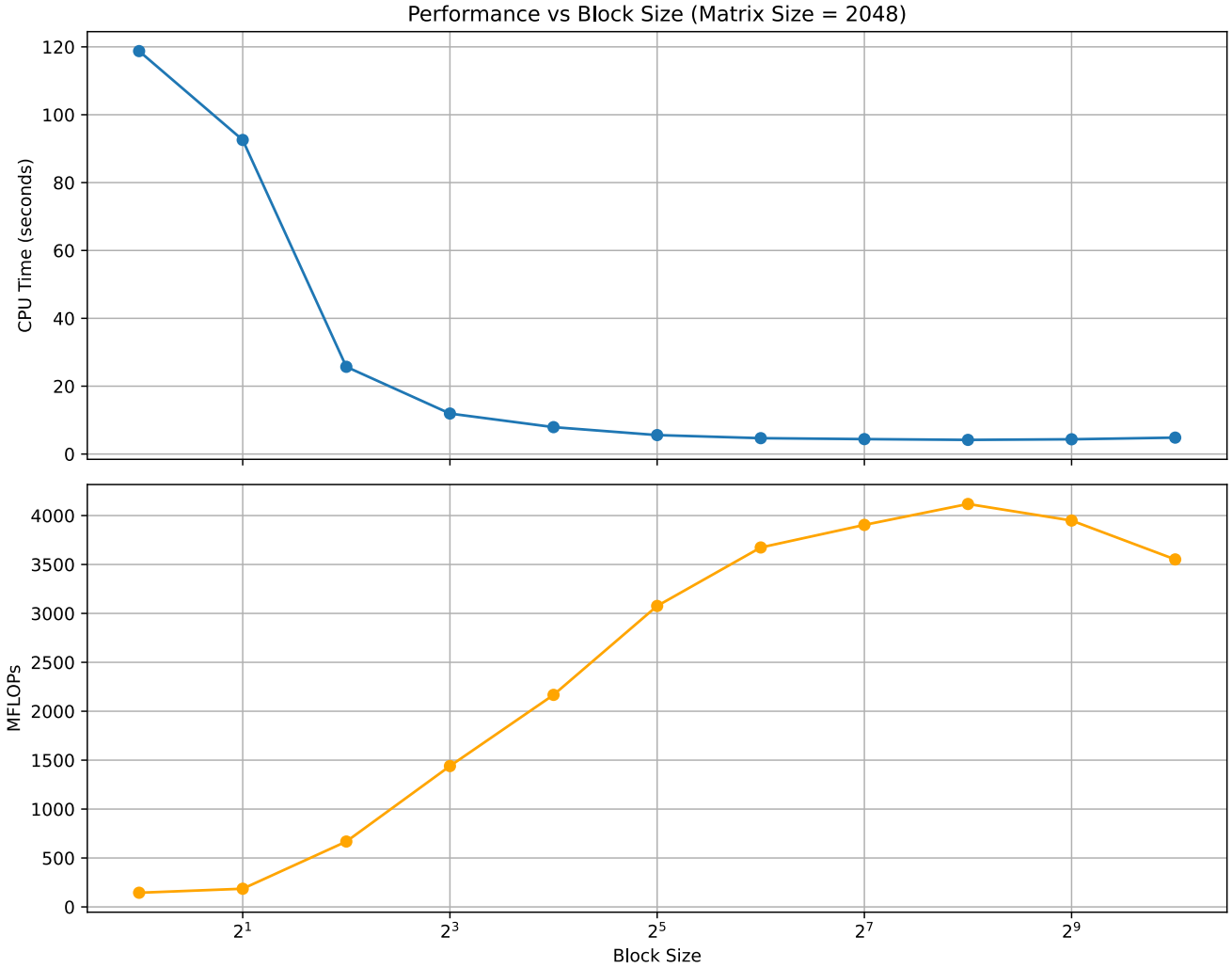


Figure 2: Performance vs Block Size (Matrix Size = 2048)

7. **Parallélisation du produit matrice–matrice par bloc** : À l'aide d'OpenMP, parallélisez le produit matrice–matrice par bloc puis mesurez l'accélération parallèle en fonction du nombre de threads. Comparez avec la version scalaire parallélisée. Comment expliquez vous ce résultat ?

   Pour la prochaine étape de parallélisation, nous allons utiliser le code suivant :

Listing 2: Parallélisation avec OpenMP

```
#pragma omp parallel
{
  #pragma omp for collapse(2) schedule(dynamic)
  for (int iRowBlkA = 0; iRowBlkA < A.nbRows; iRowBlkA += szBlock)
    for (int iColBlkB = 0; iColBlkB < B.nbCols; iColBlkB += szBlock)
      for (int iColBlkA = 0; iColBlkA < A.nbCols; iColBlkA += szBlock)
        prodSubBlocks(iRowBlkA, iColBlkB, iColBlkA, szBlock, A, B, C);
}
```

   Ce code utilise la directive `#pragma omp parallel` pour créer une région parallèle, et la directive `#pragma omp for collapse(2) schedule(dynamic)` pour paralléliser les deux premières boucles imbriquées. Cette approche permet de combiner les itérations des deux boucles afin de répartir plus efficacement la charge de travail entre les threads. En assignant à chaque thread un sous-bloc indépendant de la matrice, on optimise l'utilisation du cache en évitant de charger plusieurs fois les mêmes données dans différents caches. Ainsi, chaque processeur gère des portions spécifiques de la mémoire, ce qui améliore globalement la performance du calcul en réduisant les accès redondants en mémoire.

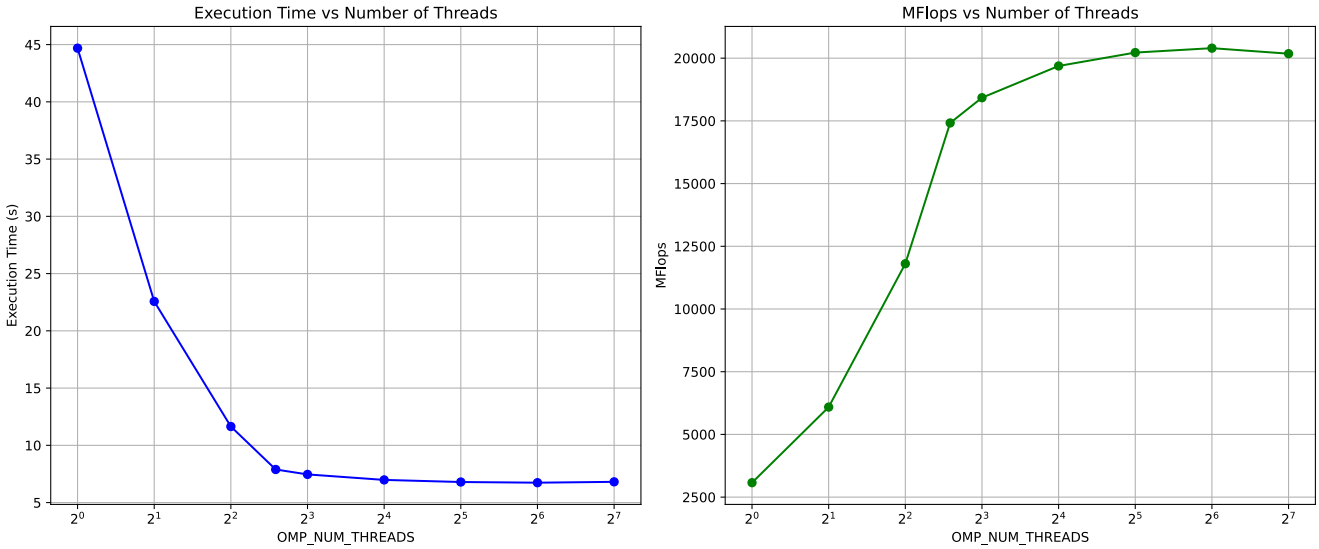


Figure 3: Speedup of matrix multiplication using OpenMP with varying thread counts.

8. **Comparaison avec blas**: Comparez vos résultat avec l'exécutable `produit_matmat_blas.exe` qui utilise un produit matrice-matrice optimisé. Quel rapport de temps obtenez vous ? Quelle version est la meilleure selon vous ?

   Réponse:

   Notre multiplication par blocs optimisée a atteint une performance maximale d'environ 20000 MFLOPS pour une matrice de taille 2048, tandis que l'implémentation BLAS réalise le même calcul en obtenant environ 219430 MFLOPS. Cela correspond à un gain d'environ 11 fois en faveur de BLAS, ce qui démontre que, malgré les améliorations significatives apportées par l'algorithme par blocs (notamment grâce à une meilleure exploitation du cache), l'optimisation poussée et les techniques de parallélisme, vectorisation et gestion mémoire de BLAS en font la version la plus performante pour la multiplication matricielle.

2 Parallélisation MPI

Ecrivez en langage C les programmes suivants.

2.1 Circulation d'un jeton dans un anneau

Ecrivez un programme tel que :

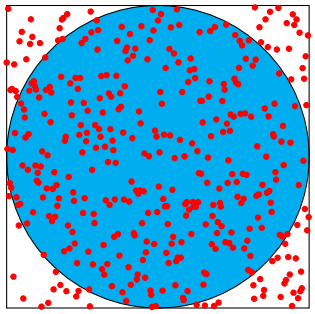
1. le processus de rang zéro initialise un jeton à 1 puis l'envoie au processus de rang un;
2. le processus de rang un reçoit le jeton, l'incrémente de un puis l'envoie au processus de rang deux;
3. ...
4. le processus de rang p reçoit le jeton, l'incrémente de un puis l'envoie au processus de rang $p+1$ $(0 < p < nbp - 1)$;
5. ...
6. le processus de rang nbp $-$ 1 reçoit le jeton, l'incrémente de un et l'envoie au processus de rang zéro;
7. le processus de rang zéro reçoit le jeton du processus nbp $-$ 1 et l'affiche à l'écran.

2.2 Calcul très approché de pi

On veut calculer la valeur de pi à l'aide de l'algorithme stochastique suivant :

- on considère le carré unité $[-1;1] \times [-1;1]$ dans lequel on inscrit le cercle unité de centre $(0,0)$ et de rayon 1;
- on génère des points aléatoirement dans le carré unité;
- on compte le nombre de points générés dans le carré qui sont aussi dans le cercle;
- soit r ce nombre de points dans le cercle divisé par le nombre de points total dans le carré, on calcule alors pi comme $\pi = 4.r$.

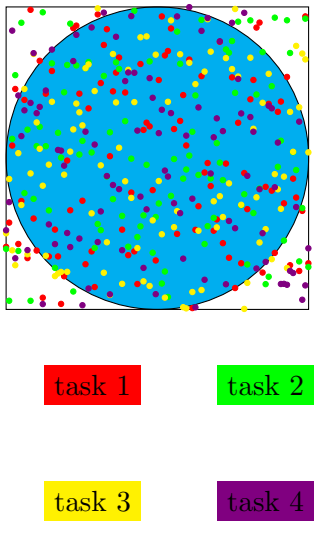
Remarquez que l'erreur faite sur pi décroit quand le nombre de points générés augmente.



Couper l'itération de boucle en plusieurs morceaux pouvant être exécutés par différentes tâches simultanément:

- chaque tâche exécute sa portion de boucle;
- chaque tâche peut exécuter son travail sans avoir besoin d'information des autres tâches (indépendance des données);
- la tâche maître (que le programmeur aura choisi parmi ses tâches) reçoit le résultat des autres tâches à l'aide d'échanges de message point à point;
- utiliser l'échange de message global adéquat pour obtenir le résultat final.

Mesurez le temps mis par les deux versions (séquentiel et parallèle) et calculez l'accélération obtenue. Est-ce cohérent avec votre machine (cf. `lscpu` ou gestionnaire de tâches) ?



Le concept : diviser le travail parmi les tâches disponibles en communicant des données à l'aide d'appel à des fonctions d'envoi/réception point à point.

A FAIRE :

- Paralléliser en mémoire partagée le programme séquentiel en C à l'aide d'OpenMP

Listing 3: Monte Carlo et OpenMP

```
double approximate_pi(unsigned long nbSamples) {
unsigned long nbDarts = 0;

#pragma omp parallel
{
unsigned int seed = static_cast<unsigned int>(
        chrono::high_resolution_clock::now().time_since_epoch().count()
                + omp_get_thread_num()
);
std::default_random_engine generator(seed);
std::uniform_real_distribution<double> distribution(-1.0, 1.0);

#pragma omp for reduction(+:nbDarts)
        for (unsigned long sample = 0; sample < nbSamples; ++sample) {
                double x = distribution(generator);
                double y = distribution(generator);
                if (x * x + y * y <= 1.0)
                        nbDarts++;
        }
}

double ratio = static_cast<double>(nbDarts) / static_cast<double>(nbSamples);
return 4.0 * ratio;
}
```

- Mesurez l'accélération obtenue en utilisant un nombre variable de coeurs de calcul

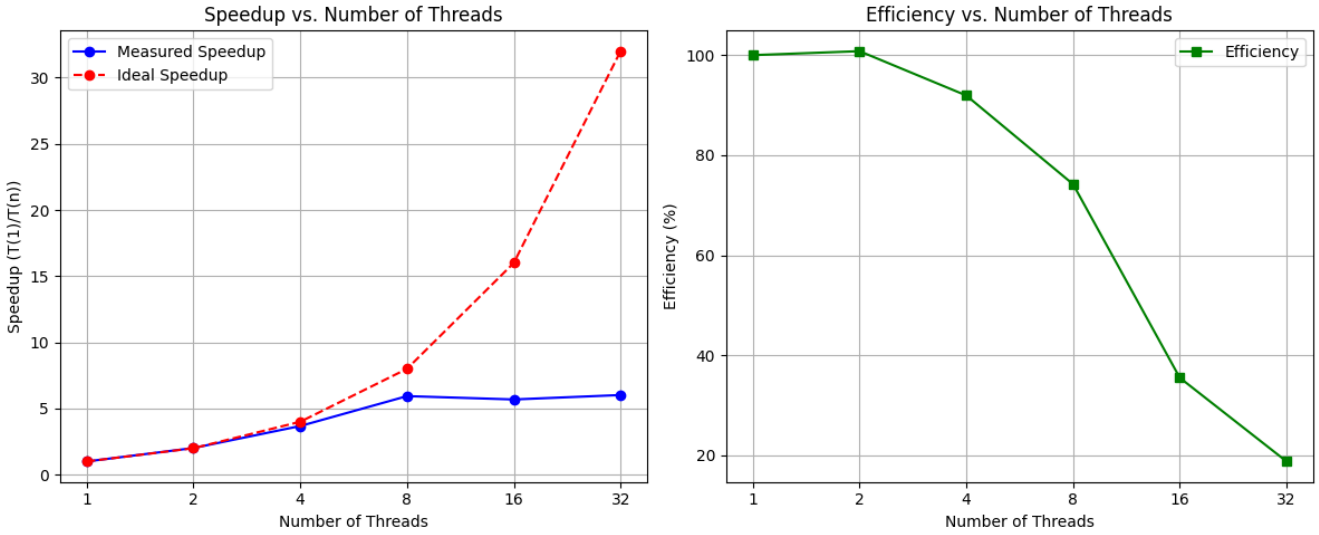


Figure 4: Monte Carlo - OpenMP with varying thread counts.

- Paralléliser en mémoire distribuée le programme séquentiel en C à l'aide de MPI

Listing 4: Monte Carlo et MPI reduction

```
int main(int argc, char* argv[])
{
    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm globComm;
    MPI_Comm_dup(MPI_COMM_WORLD, &globComm);

    // Get number of processes and current process rank
    int nbp, rank;
    MPI_Comm_size(globComm, &nbp);
    MPI_Comm_rank(globComm, &rank);

    // Distribute samples among processes
    unsigned long totalSamples = 100000000;
    if (argc > 1)
```

```
        totalSamples = std::stoul(argv[1]);
    unsigned long localSamples = totalSamples / nbp;

    // Each process performs its part of the simulation
    unsigned long localCount = 0;
    std::default_random_engine generator(static_cast<unsigned>(
        std::chrono::high_resolution_clock::now().time_since_epoch().count()) + rank);
    std::uniform_real_distribution<double> distribution(-1.0, 1.0);

    for (unsigned long i = 0; i < localSamples; ++i) {
        double x = distribution(generator);
        double y = distribution(generator);
        if (x * x + y * y <= 1.0)
            localCount++;
    }

    // Reduce local counts to compute the total count
    unsigned long globalCount = 0;
    MPI_Reduce(&localCount, &globalCount, 1, MPI_UNSIGNED_LONG, MPI_SUM, 0, globComm);

    // The root process computes and prints the final result
    if (rank == 0) {
        unsigned long usedSamples = localSamples * nbp;
        double pi = 4.0 * static_cast<double>(globalCount) / static_cast<double>(usedSamples);
        std::cout << "Approximation de pi = " << pi << std::endl;
    }

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

- Mesurez l'accélération obtenue en utilisant un nombre variable de processus

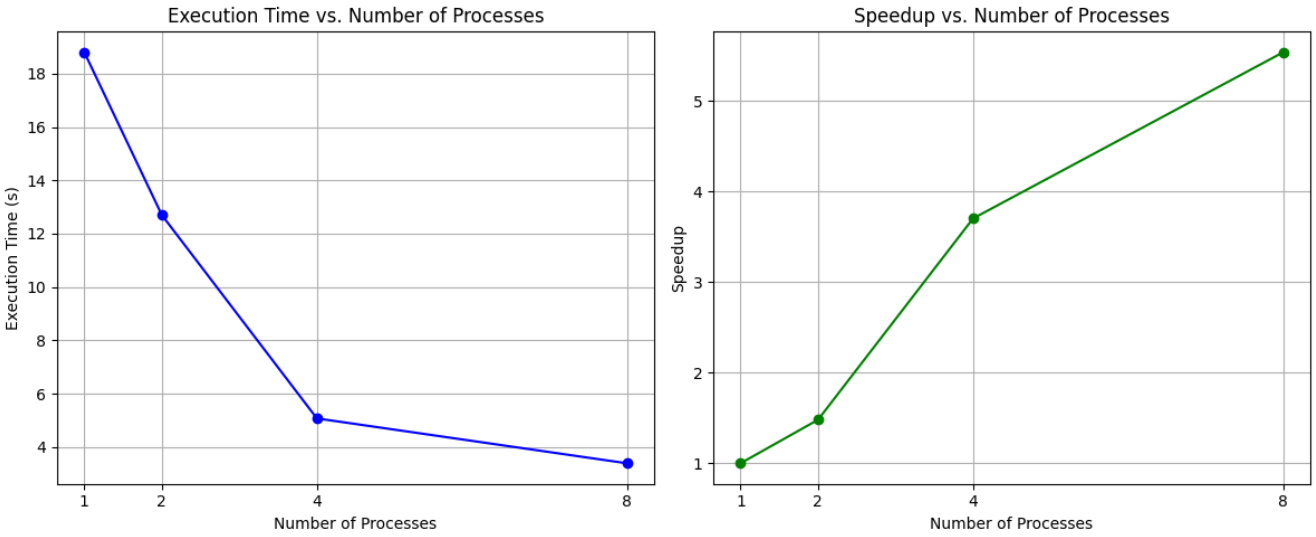


Figure 5: Monte Carlo - MPI with varying thread counts.

- Paralléliser en mémoire distribuée le programme séquentiel en Python à l'aide de mpi4py

Listing 5: Monte Carlo et MPI4PY reduction

```
from mpi4py import MPI
import numpy as np
import time

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

nb_samples_total = 100_000_000

samples_per_process = nb_samples_total // size
reste = nb_samples_total % size
if rank < reste:
    samples_per_process += 1
```

```python
if rank == 0:
    start_time = time.time()

x = 2.0 * np.random.random_sample(samples_per_process) - 1.0
y = 2.0 * np.random.random_sample(samples_per_process) - 1.0

local_count = np.sum(x*x + y*y < 1)

global_count = comm.reduce(local_count, op=MPI.SUM, root=0)

if rank == 0:
    approx_pi = 4.0 * global_count / nb_samples_total
    end_time = time.time()
    print(f"Temps pour calculer pi : {end_time - start_time} secondes")
    print(f"Pi vaut environ {approx_pi}")
```

- Mesurez l'accélération obtenue en utilisant un nombre variable de processus

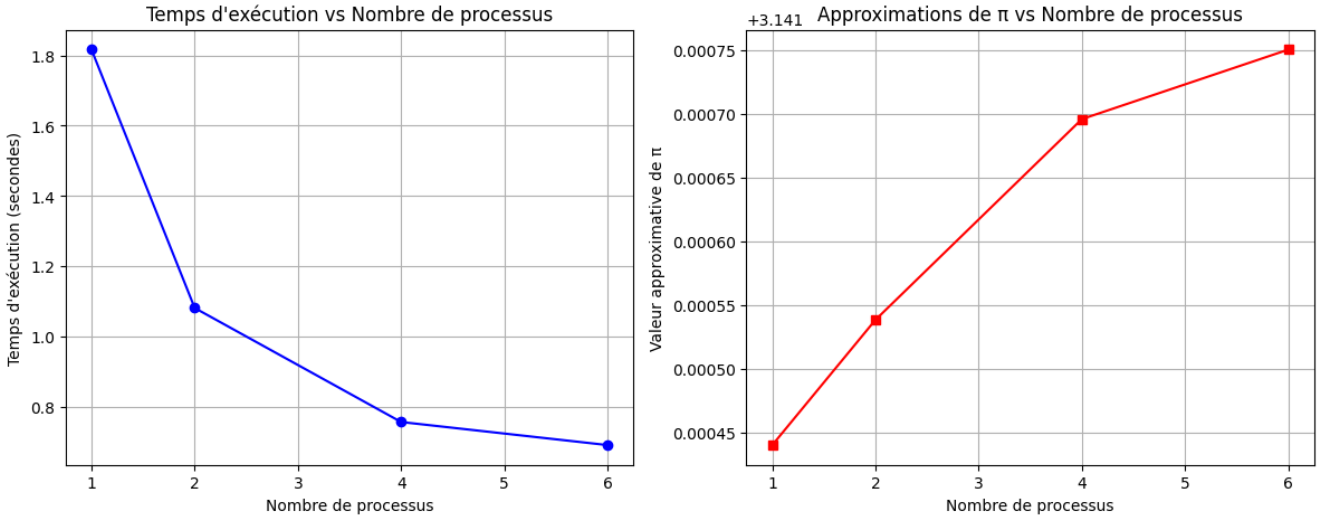
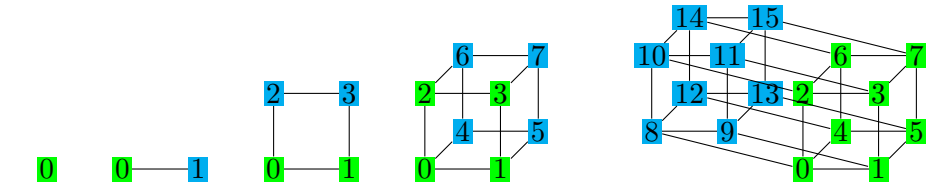


Figure 6: Monte Carlo - MPI4PY with varying thread counts.

2.3 Diffusion d'un entier dans un réseau hypercube*

On veut écrire un programme qui diffuse un entier dans un réseau de nœuds de calculs dont la topologie est équivalente à celle d'un hypercube de dimension d (et qui contient donc 2^d nœuds de calcul).

1. Écrire un programme en C qui diffuse un entier dans un hyper cube de dimension 1 :
   - la tâche 0 initialise un jeton à une valeur entière choisie par le programmeur et envoie cette valeur à la tâche 1;
   - la tâche 1 reçoit la valeur du jeton de la tâche 0;
   - les deux tâches affichent la valeur du jeton.
2. Diffuser le jeton généré par la tâche 0 dans un hypercube de dimension 2 de manière que cette diffusion se fasse en un minimum d'étapes (et donc un maximum de communications simultanées entre tâches).
3. Faire de même pour un hypercube de dimension 3.
4. Écrire le cas général quand le cube est de dimension d. Le nombre d'étapes pour diffuser le jeton devra être égal à la dimension de l'hypercube.
5. Mesurer l'accélération obtenue pour la diffusion d'un entier sur un tel réseau.



dim. 0 dim. 1 dim. 2 dim. 3 dim. 4

Hypercubes de dimension 0, 1, 2, 3 et 4