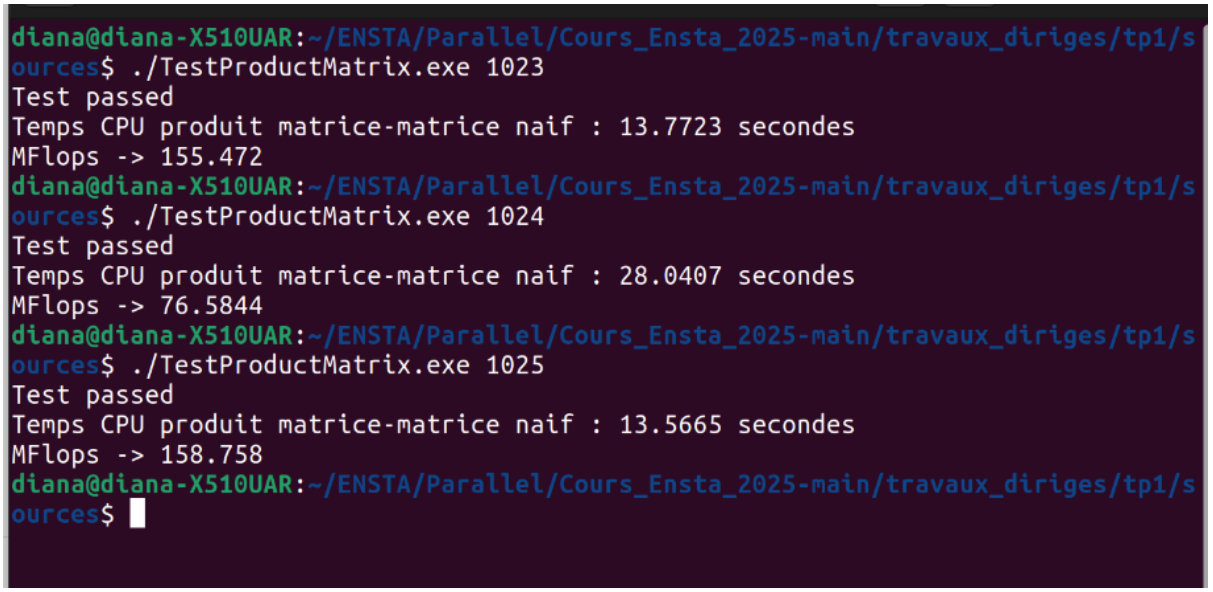
Systèmes parallèles et distribués

Travaux dirigés n°1

Diana Tymoshenko

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 sufft 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.



Dimesnion	Sec
1023	13.7723
1024	28.0407
1025	13.5665

La dimension 1024 montre des performances inférieures par rapport à 1023 ou 1025.

Avec 1024, les données sont parfaitement alignées en mémoire. Mais cela signifie également que les lignes de cache sont réécrites plus souvent, provoquant des conflits de cache (cache thrashing).
Les tailles 1023 et 1025 décalent l'accès à la mémoire, réduisant ainsi les conflits dans le cache.

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.

Boucle	Sec
i, k, j	28.0407
i, j, k	4.59673
j, i, k	6.19742
j, k, i	1.11357
k, i, j	19.7661
k, j, i	0.866741

Meilleur résultat pour les boucles k, j, i et j, k, i, et le pire pour i, k, j.

Cela est lié à la méthode de stockage des données dans la matrice : la classe Matrix utilise un ordre de stockage par colonnes.
Cela signifie :

- Les éléments d'une même colonne sont stockés de manière contiguë en mémoire.
- Les éléments d'une même ligne sont séparés par un grand pas (nbRows).

Dans les boucles k, j, i et j, k, i, la lecture des matrices se fait principalement par colonnes, ce qui correspond à un accès efficace pour le cache.
Alors que dans la boucle i, k, j, l'accès à tous les éléments A(i, k), B(k, j) et C(i, j) se fait par lignes, ce qui est inefficace pour un stockage orienté colonnes.
Cela entraîne des modifications fréquentes de différents éléments en mémoire, augmentant ainsi les ratés de cache (cache misses) et, par conséquent, les accès à la RAM.

```
for (int i = 0; i < N; i++) {
   for (int k = 0; k < N; k++) {
      for (int j = 0; j < N; j++) {
         C(i, j) += A(i, k) * B(k, j);
      }
   }
}
```

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.

On utilise la boucle j, k, i.

```
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
OMP_NUM_THREADS=1 ./TestProductMatrix.exe
Test passed
Temps CPU produit matrice-matrice naif : 0.831105 secondes
MFlops -> 2583.89
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
OMP_NUM_THREADS=2 ./TestProductMatrix.exe
Test passed
Temps CPU produit matrice-matrice naif : 0.429763 secondes
MFlops -> 4996.91
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
OMP_NUM_THREADS=4 ./TestProductMatrix.exe
Test passed
Temps CPU produit matrice-matrice naif : 0.215387 secondes
MFlops -> 9970.37
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
OMP_NUM_THREADS=6 ./TestProductMatrix.exe
Test passed
Temps CPU produit matrice-matrice naif : 0.237479 secondes
MFlops -> 9042.85
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
OMP_NUM_THREADS=8 ./TestProductMatrix.exe
Test passed
Temps CPU produit matrice-matrice naif : 0.189285 secondes
MFlops -> 11345.3
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
OMP_NUM_THREADS=16 ./TestProductMatrix.exe
Test passed
Temps CPU produit matrice-matrice naif : 0.194014 secondes
MFlops -> 11068.7
```

OMP_NUM_THREADS	Sec
1	0.831105
2	0.429763
4	0.215387
8	0.189285
16	0.194014

Le pire résultat est obtenu avec 1 thread, le meilleur avec 8 threads.
L'efficacité optimale dépend de l'architecture du processeur. Dans mon cas, il s'agit de 4 cœurs physiques (8 logiques).

Avec 1 thread : le résultat est le plus lent, car les capacités du processeur multicœur ne sont pas utilisées.

Avec 2, 4, 8 threads : il y a une augmentation des performances, car OpenMP utilise efficacement plusieurs cœurs, ce qui permet une exécution plus rapide.

16 threads dépassent le nombre de cœurs logiques disponibles. Les threads commencent à se concurrencer pour les mêmes ressources CPU, ce qui entraîne :

- Des commutations de contexte (context switching) — des opérations coûteuses qui augmentent les surcharges.
- Une perte d'efficacité du cache — plus de threads = plus de ratés de cache (cache misses), car les données sont remplacées plus fréquemment.
- Des surcharges liées à la gestion des threads supplémentaires par OpenMP.

Il n'est pas conseillé d'utiliser plus de threads que de cœurs logiques — cela dégrade les performances en raison de la concurrence pour les ressources.

---

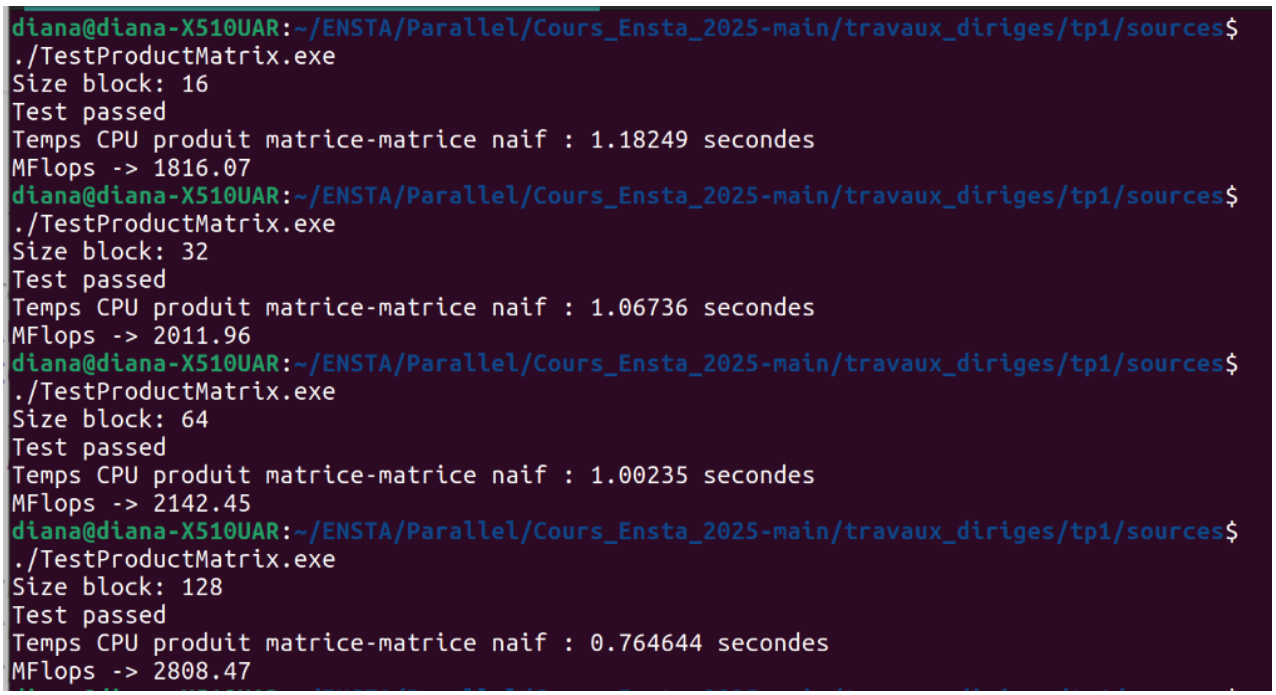
4. Argumentez et donnez clairement la raison pour laquelle il est sûrement possible d'améliorer le résultat que vous avez obtenu.

On peut également implémenter la multiplication de matrices par blocs, au lieu de lire des lignes et des colonnes entières. Cela améliorera l'utilisation de la mémoire cache.

Par exemple, en choisissant une taille de bloc optimale pour la mémoire cache L1/L2 du processeur, on réduit le nombre de ratés de cache (cache misses), car les sous-blocs tiennent facilement dans le cache L1. Cela devrait améliorer les performances.

---

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).



Size block	Sec
16	1.18249
32	1.06736
64	1.00235
128	0.764644

Avec une petite taille de bloc (16), le processeur passe souvent du temps à accéder à la mémoire RAM.

Lorsque la taille du bloc augmente à 32, 64 et surtout 128, le processeur utilise mieux le cache L1/L2, car :

- Les données sont lues et traitées dans un seul bloc de cache.
- Il y a moins d'accès à la RAM, qui est plus lente.

Si le bloc est trop grand, différentes parties du programme entrent en concurrence pour la mémoire cache.
Les données encore nécessaires aux calculs peuvent être "évincées" du cache, ce qui entraîne des accès supplémentaires à la RAM, plus lente.

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

”scalaire”	”bloc–bloc”
1.11357	0.764644

Lors de l'utilisation d'un schéma par blocs, les données pour les calculs sont lues et traitées par parties, ce qui permet de réduire le nombre d'accès à la RAM plus lent.

Les blocs plus petits augmentent la probabilité que les éléments nécessaires des matrices soient déjà présents dans le cache, ce qui réduit le temps d'accès à la mémoire.

La multiplication par blocs permet de réduire le nombre d'opérations et les coûts temporaires associés à l'accès à la mémoire et à l'échange de données entre les cœurs du processeur.

---

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 ?

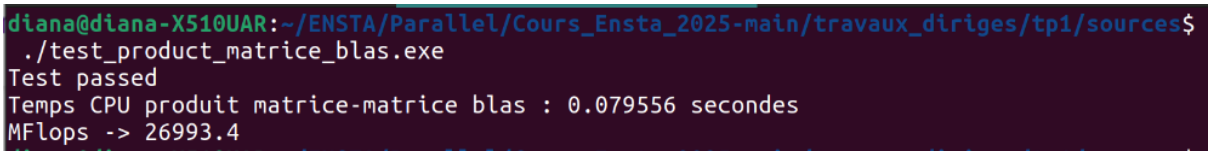
```
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=1 ./TestProductMatrix.exe
Size block: 128
Test passed
Temps CPU produit matrice-matrice naif : 0.756748 secondes
MFlops -> 2837.78
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=2 ./TestProductMatrix.exe
Size block: 128
Test passed
Temps CPU produit matrice-matrice naif : 0.386419 secondes
MFlops -> 5557.4
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=4 ./TestProductMatrix.exe
Size block: 128
Test passed
Temps CPU produit matrice-matrice naif : 0.203507 secondes
MFlops -> 10552.4
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=8 ./TestProductMatrix.exe
Size block: 128
Test passed
Temps CPU produit matrice-matrice naif : 0.194792 secondes
MFlops -> 11024.5
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=16 ./TestProductMatrix.exe
Size block: 128
Test passed
Temps CPU produit matrice-matrice naif : 0.195585 secondes
MFlops -> 10979.8
```

OMP_NUM_THREADS	Sec (par block)	Sec (scalaire)
1	0.756748	0.831105
2	0.386419	0.429763
4	0.203507	0.215387
8	0.194792	0.189285
16	0.195585	0.194014

Le produit par blocs est plus performant que la version scalaire, surtout pour un faible nombre de threads, grâce à sa meilleure gestion des données en mémoire.

---

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 ?



0.194014 / 0.079566 = 2.4384

Cela signifie que blas est environ 2.4 fois plus rapide que l'implémentation par blocs parallèle avec OpenMP, blas est plus performant.

2. Parallélisation MPI

Ecrivez en langage C les programmes suivants.

2.1 Circulation d'un jeton dans un anneau

anneau_mpi.cpp

```
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 mpirun -np 4 anneau_mpi
Process 0 initialize token: 1
Process 1 received token: 1
Process 2 received token: 2
Process 3 received token: 3
Process 0 received token: 4
```

2.2 Calcul très approché de pi

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

pi_openmp.cpp

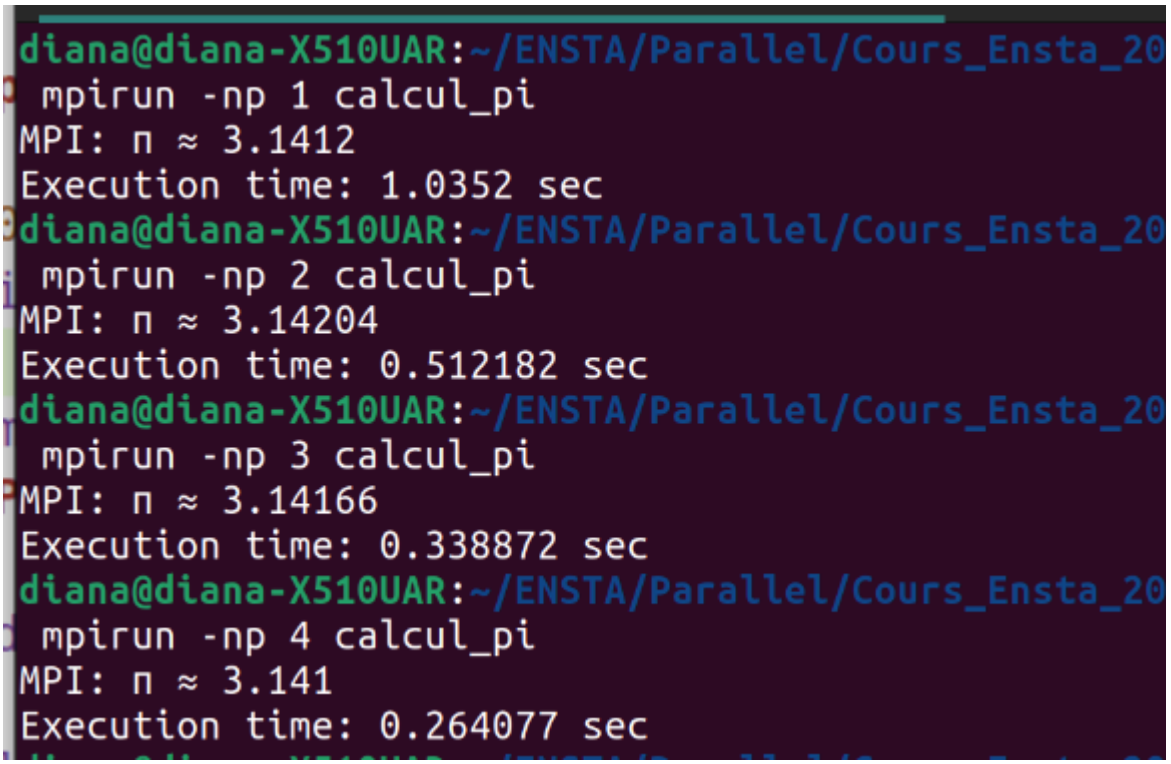
```
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=1 ./pi_openmp
OpenMP: π ≈ 3.1416780000
Extcution time: 0.17726 sec
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=2 ./pi_openmp
OpenMP: π ≈ 3.1415880000
Extcution time: 0.08896 sec
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=4 ./pi_openmp
OpenMP: π ≈ 3.1416960000
Extcution time: 0.04893 sec
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=8 ./pi_openmp
OpenMP: π ≈ 3.1421828000
Extcution time: 0.03925 sec
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 OMP_NUM_THREADS=16 ./pi_openmp
OpenMP: π ≈ 3.1418688000
Extcution time: 0.03864 sec
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
```

n = 10 000 000

OMP_NUM_THREADS	Sec	Pi
1	0.17726	3.141678
2	0.08896	3.141588
4	0.04893	3.141696
8	0.03925	3.1421828
16	0.03864	3.1418688

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

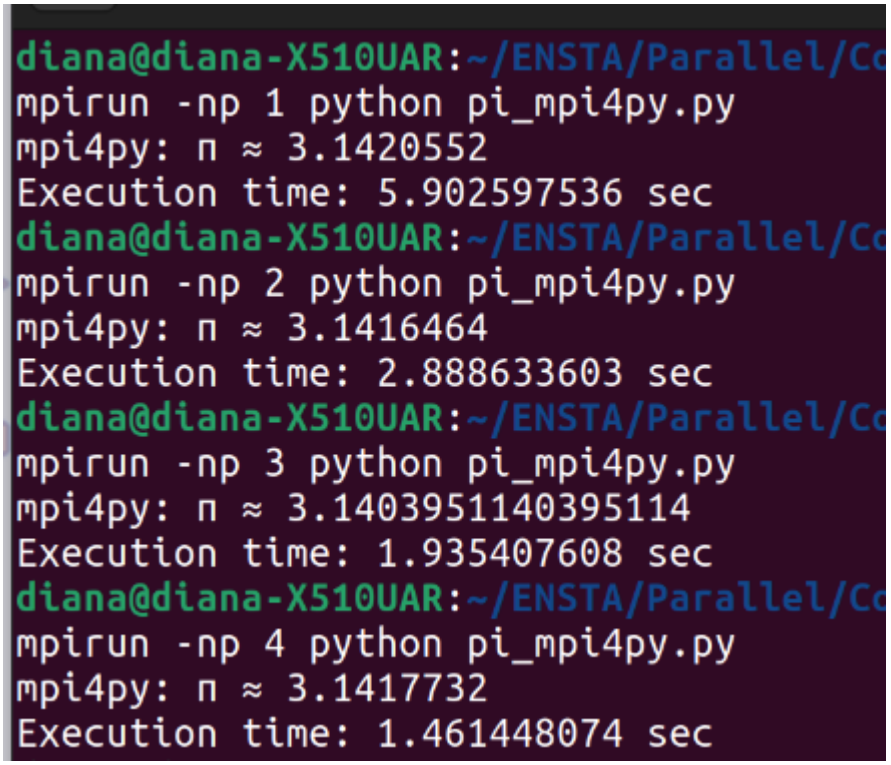
calcul_pi.cpp



n = 10 000 000

Processes	Sec	Pi
1	1.0352	3.1412
2	0.512182	3.14204
3	0.338872	3.14166
4	0.264077	3.141

- Paralléliser en mémoire distribuée le programme séquentiel en Python à l'aide de mpi4py
  pi_mpi4py.py

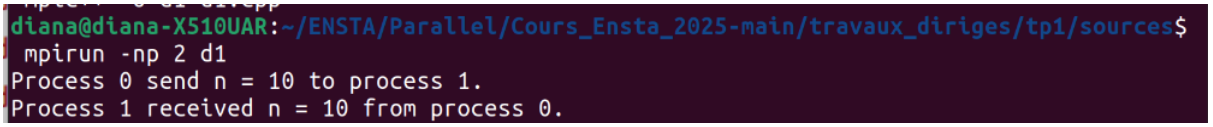


n = 10 000 000

Processes	Sec	Pi
1	5.9026	3.1420552
2	2.8886	3.1416464
3	1.9354	3.1403951
4	1.4614	3.1417732

2.3 Diffusion d'un entier dans un réseau hypercube⋆

- Écrire un programme en C qui diffuse un entier dans un hypercube de dimension 1
  d1.cpp



- 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.
  d2.cpp

```
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
 mpirun -np 4 d2
Process 0 send n = 10 to process 1.
Process 0 send n = 10 to process 2.
Process 2 received n = 10 from process 0.
Process 2 send n = 10 to process 3.
Process 1 received n = 10 from process 0.
Process 1 send n = 10 to process 3.
Process 3 received n = 10 from processes 1 and 2.
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
```

- Faire de même pour un hypercube de dimension 3. 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.

d3.cpp

```
diana@diana-X510UAR:~/ENSTA/Parallel/Cours_Ensta_2025-main/travaux_diriges/tp1/sources$
mpirun --oversubscribe -np 8 ./d3
Process 0 set M = 10
Process 0 send n = 10 to process 4.
Process 0 send n = 10 to process 2.
Process 0 send n = 10 to process 1.
Process 1 received M = 10 from process 0.
Process 2 received M = 10 from process 0.
Process 2 send n = 10 to process 3.
Process 4 received M = 10 from process 0.
Process 4 send n = 10 to process 6.
Process 4 send n = 10 to process 5.
Process 3 received M = 10 from process 2.
Process 5 received M = 10 from process 4.
Process 6 received M = 10 from process 4.
Process 6 send n = 10 to process 7.
Process 7 received M = 10 from process 6.
Total time: 0.000518331 seconds.
```