2024/25

Nom : DADA SIMEU CÉDRIC DAREL

Email : cedric-darel.dada@ensta-paris.fr

Titre : Compte rendu TP2

STIC

ENSTA Paris, Institut Polytechnique de Paris

Table des matières

Table des figures

1 Produit matrice-matrice

1.1 Explication des résultats

Dimension	Temps CPU (s)	MFlops
1023	12.5931	170.03
1024	17.6935	121.371
1025	12.8699	167.35

TABLE 1 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices.

Lorsque l'on exécute le programme avec différentes dimensions (1023, 1024, et 1025), on constate que : La dimension 1024 prend plus de temps que les autres. Cela est dû à l'alignement des données dans la mémoire cache. Lorsque la taille de la matrice correspond exactement à une puissance de deux (comme 1024), les lignes successives de la matrice peuvent se retrouver sur les mêmes lignes de cache, provoquant des conflits de cache (cache thrashing). Cela force le système à recharger fréquemment les lignes de cache, augmentant ainsi le temps d'exécution.

Solution proposée : Pour résoudre ce problème, on peut ajouter un padding (remplissage) aux matrices afin de décaler leur alignement en mémoire. Par exemple, ajoutez une ligne ou une colonne supplémentaire pour éviter ces conflits.

1.2 Première optimisation : Permutez les boucles jusqu'à obtenir un temps optimum

Dans prodSubBlocks, les boucles sont organisées comme suit :

Dimension	Temps CPU (s)	MFlops
1023	12.5931	170.03
1024	17.6935	121.371
1025	12.8699	167.35

TABLE 2 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices (i,k,j).

Dimension	Temps CPU (s)	MFlops
1023	1.46737	1459.21
1024	7.05546	304.372
1025	1.52632	1411.1

TABLE 3 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices (permutation i,j,k).

Dimension	Temps CPU (s)	MFlops
1023	1.12464	1915.09
1024	1.1314	1892.53
1025	1.1196	1918.09

TABLE 4 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices (permutation j,k,i).

Dimension	Temps CPU (s)	MFlops
1023	3.52183	584.697
1024	7.20673	297.983
1025	3.68359	1918.09

TABLE 5 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices (permutation j,i,k).

Dimension	Temps CPU (s)	MFlops
1023	10.2218	209.474
1024	17.5939	122.059
1025	10.2415	1918.09

TABLE 6 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices (permutation k,i,j).

Dimension	Temps CPU (s)	MFlops
1023	1.19066	1798.32
1024	1.03645	2071.97
1025	1.66156	1296.24

TABLE 7 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices (permutation k,j,i).

Optimisation : Le stockage des matrices dans Matrix est basé sur un format colonne-major (car m_arr_coefs[i+j*nbRows] accède d'abord aux colonnes). Pour maximiser l'utilisation du cache, il est préférable de permuter les boucles afin de parcourir les colonnes avant les lignes. La permutation optimale serait :

```
        for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); j++)
            for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); k++)
                for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
                        C(i, j) += A(i, k) * B(k, j);


```

Explication :

En parcourant les colonnes avant les lignes, nous minimisons les sauts en mémoire et exploitons pleinement les données chargées dans le cache. En effet, lorsqu'une donnée est chargée dans le cache, quelques unes de ses valeurs voisines y sont également chargées, et ainsi, en permutant la boucle de cette façon, nous récupérons ces données directement dans le cache. Cette permutation réduit significativement le temps d'exécution. Ainsi nous commençons par itérer sur j car j désigne toujours une colonne dans le calcul C(i, j) += A(i, k) * B(k, j). Ensuite nous continuons par k qui désigne en meme temps une ligne dans B et une colonne dans A. Nous terminons ensuite par i qui désigne une ligne dans A.

1.3 Première parallélisation

À l'aide d'OpenMP, parallélisons le produit matrice–matrice

```
        void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
        const Matrix& A, const Matrix& B, Matrix& C) {
          #pragma omp parallel for
          for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); j++) {
              for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); k++) {
                  for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock); ++i)
    {
                      C(i, j) += A(i, k) * B(k, j);
                  }
              }
          }
      }

```

Explication

— La directive #pragma omp parallel for nous permet de paralléliser la boucle externe sur j, distribuant les colonnes de C entre les threads.

— Ainsi, chaque thread calcule indépendamment les contributions pour ses colonnes attribuées, garantissant l'absence de conflits d'accès mémoire

Mesure de l'acclération .

Pour OMP_NUM_THREADS=4, on a une accélaration de $\frac{1,537}{0.512} = 3.00$

Pour OMP_NUM_THREADS=16, on a une accélaration de $\frac{1,537}{0.4045} = 3.80$

Dimension	Temps CPU (s)	MFlops
1023	1.30567	1639.92
1024	1.87234	1146.95
1025	1.43565	1500.21
Moyenne	1,537	1429,03

TABLE 8 – Performances pour OMP_NUM_THREADS=1.

Dimension	Temps CPU (s)	MFlops
1023	0.384401	5570.22
1024	0.68271	3145.53
1025	0.46899	4592.38
Moyenne	0,512	4436,04

TABLE 9 – Performances pour OMP_NUM_THREADS=4.

Dimension	Temps CPU (s)	MFlops
1023	0.358636	5970.4
1024	0.431726	4974.18
1025	0.423238	5088.82
Moyenne	0,4045	5344,47

TABLE 10 – Performances pour OMP_NUM_THREADS=16.

1.4 Pourquoi il est possible d'améliorer encore les résultats ?

Les performances peuvent être améliorées grâce au blocking (ou blocage de cache). Diviser les matrices en sous-blocs permet de mieux exploiter les caches L1/L2.

1.5 Deuxième optimisation : Optimisez le produit matrice–matrice par bloc

Code :

```
        namespace {
            void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
                               const Matrix& A, const Matrix& B, Matrix& C) {
                for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); ++j) {
                    for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); ++
    k) {
                        const double b_kj = B(k, j);
                        for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock)
    ; ++i) {
                            C(i, j) += A(i, k) * b_kj;
                        }
                    }
                }
            }

            int findOptimalBlockSize(const Matrix& A, const Matrix& B) {
                std::vector<int> sizes = {16, 32, 64, 128, 256};
                double best_time = 1e9;
                int best_size = 64;

                for (int sz : sizes) {
                    Matrix C(A.nbRows, B.nbCols, 0.0);
                    double start = omp_get_wtime();

                    for (int I = 0; I < A.nbRows; I += sz) {
                        for (int J = 0; J < B.nbCols; J += sz) {
                            for (int K = 0; K < A.nbCols; K += sz) {
                                prodSubBlocks(I, J, K, sz, A, B, C);
                            }
                        }
                    }

                    double duration = omp_get_wtime() - start;
                    if (duration < best_time) {
                        best_time = duration;
                        best_size = sz;
                    }
                }

                std::cout << "[OPTIM] Taille de bloc optimale: " << best_size << std::
    endl;
                return best_size;
            }
            } // namespace

            Matrix operator*(const Matrix& A, const Matrix& B) {
```

```
44                  Matrix C(A.nbRows, B.nbCols, 0.0);
45                  const int szBlock = 256;
46
47                  for (int I = 0; I < A.nbRows; I += szBlock) {
48                      for (int J = 0; J < B.nbCols; J += szBlock) {
49                          for (int K = 0; K < A.nbCols; K += szBlock) {
50                              prodSubBlocks(I, J, K, szBlock, A, B, C);
51                          }
52                      }
53                  }
54                  return C;
55              }
```

TABLE 11 – Temps d'exécution (secondes) pour N=1023–1025 et 2046–2050

Taille	16	32	64	128	256	512	1024	2048
1023	0.793446	0.665258	0.635098	0.637810	0.573094	**0.561053**	0.725468	0.732480
1024	0.886341	0.681334	0.628902	0.635453	0.567116	**0.552188**	0.706114	0.715358
1025	0.801337	0.685229	0.630492	0.639878	**0.574363**	0.579971	0.733085	0.739268
2046	6.439210	5.387410	4.786580	4.901310	4.598490	**4.359200**	6.310960	6.181470
2048	8.126130	5.874090	5.053240	4.902760	**4.416660**	4.432710	6.007550	6.248670
2050	6.606900	5.658080	5.137750	5.149390	4.777270	**4.640320**	6.353000	6.168310

Conclusion : Nous voyons que globalement, la taille optimale est **512**.

Commentaire : Notre machine possède 6MiB de cache L3 et 1MiB répartis sur 4 cœurs en cache L2, soit environ 256KiB par cœur (voir fig 2 et 1). La taille 512 est optimale pour les raisons suivantes :

— Lors du calcul de l'élément $C_{IJ} = (A_{I1} \dots A_{IN}) \times (B_{1J} \dots B_{NJ})^t$, nous avons besoin de sommer des produits de la forme $A_{IK}B_{KJ}$ et les stocker dans C_{IJ}. Le calcul de ces produits pouvant se faire indépendamment, nous avons donc besoin à la limite d'avoir les 3 blocs C_{IJ}, A_{IK} et B_{KJ} dans le cache. Nous avons donc besoin de $3 \times 512 \times 512 \times \text{sizeof(double)} = 3 \times 512 \times 512 \times 8 = 6291456$ octets $= 6144KB =$ taille cache L3(voir fig 2)

— Notons également que dans certains cas, la taille de bloc 256 est optimale. En effet, un bloc de taille 256 occupe $256 \times 256 \times 8 = 524288$octets $= 512$KiB. Or le cache L2 occupe au totale 1MiB et donc permet de stocker 2 blocs de matrice. Les processus concurents peuvent alors effectuer le produit de deux blocs à partir du cache.

```
cedric@ns2:/media/cedric/DSCD/Notes cours 2A/Parallel_architecture/Cours_Ensta_2025/travaux_diriges/tp1/sources$ lscpu
Architecture :                            x86_64
  Mode(s) opératoire(s) des processeurs :  32-bit, 64-bit
  Address sizes:                          39 bits physical, 48 bits virtual
  Boutisme :                              Little Endian
Processeur(s) :                           8
  Liste de processeur(s) en ligne :       0-7
Identifiant constructeur :                GenuineIntel
  Nom de modèle :                         Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
    Famille de processeur :               6
    Modèle :                              142
    Thread(s) par cœur :                  2
    Cœur(s) par socket :                  4
    Socket(s) :                           1
    Révision :                            12
    Vitesse maximale du processeur en MHz : 4200,0000
    Vitesse minimale du processeur en MHz : 400,0000
    BogoMIPS :                            4199.88
    Drapaux :                             fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
                                          call nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopo
                                          clmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
                                          e_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb ssbd ibr
                                          lexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx
                                          aveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window h
                                          abilities
Virtualization features:
  Virtualisation :                        VT-x
Caches (sum of all):
  L1d:                                    128 KiB (4 instances)
  L1i:                                    128 KiB (4 instances)
  L2:                                     1 MiB (4 instances)
  L3:                                     6 MiB (1 instance)
NUMA:
  Nœud(s) NUMA :                          1
  Nœud NUMA 0 de processeur(s) :          0-7
Vulnerabilities:
  Gather data sampling:                   Mitigation; Microcode
  Itlb multihit:                          KVM: Mitigation: VMX disabled
  L1tf:                                   Not affected
  Mds:                                    Not affected
  Meltdown:                               Not affected
  Mmio stale data:                        Mitigation; Clear CPU buffers; SMT vulnerable
  Reg file data sampling:                 Not affected
  Retbleed:                               Mitigation; Enhanced IBRS
  Spec rstack overflow:                   Not affected
  Spec store bypass:                      Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:                             Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:                             Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling; PBRSB-eIBRS SW
  Srbds:                                  Mitigation; Microcode
  Tsx async abort:                        Not affected
```

FIGURE 1 – Résultat de la commande lscpu

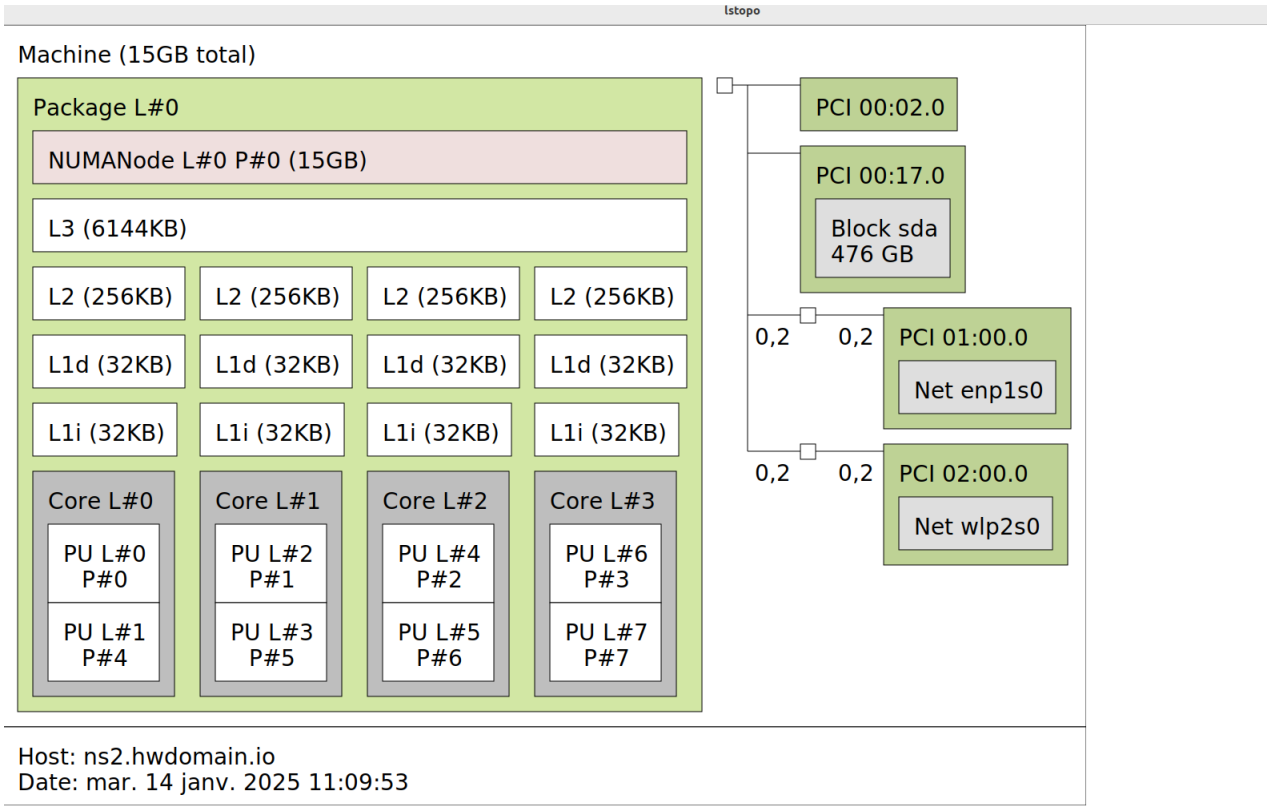

FIGURE 2 – Résultat de la commande lstopo : Nous pouvons visualiser les tailles des caches

1.6 Comparer le temps pris par rapport au produit matrice–matrice "scalaire"

La version par bloc est significativement plus rapide grâce à une meilleure utilisation des caches. Pour une taille de blocs de **512**, et un thread, nous avons une durée de moyenne (pour les dimensions 1023, 1024 et 1025) de **0,564 secondes** contre **1,537 secondes** pour le produit matrice-matrice scalaire.

1.7 Parallélisation du produit matrice–matrice par bloc

Code :

```
Matrix operator*(const Matrix& A, const Matrix& B) {
    Matrix C(A.nbRows, B.nbCols, 0.0);
    const int szBlock = findOptimalBlockSize(A, B);

    #pragma omp parallel for
    for (int I = 0; I < A.nbRows; I += szBlock) {
        for (int J = 0; J < B.nbCols; J += szBlock) {
            for (int K = 0; K < A.nbCols; K += szBlock) {
                prodSubBlocks(I, J, K, szBlock, A, B, C);
            }
        }
    }
    return C;
}
```

Vérifions si la taille de bloc 512 reste optimale en variant le nombre de threads Les figures 3, 4, 5 et 6 nous montrent que à quelques exceptions près, la taille de bloc 512 nous fournit les meilleurs temps d'exécution.

TABLE 12 – Temps d'exécution (secondes) pour N=1023–1025 et 2046–2050 (OMP_NUM_THREADS=1)

Taille	16	32	64	128	256	512	1024	2048
1023	0.793446	0.665258	0.635098	0.637810	0.573094	**0.561053**	0.725468	0.732480
1024	0.886341	0.681334	0.628902	0.635453	0.567116	**0.552188**	0.706114	0.715358
1025	0.801337	0.685229	0.630492	0.639878	**0.574363**	0.579971	0.733085	0.739268
2046	6.439210	5.387410	4.786580	4.901310	4.598490	**4.359200**	6.310960	6.181470
2048	8.126130	5.874090	5.053240	4.902760	**4.416660**	4.432710	6.007550	6.248670
2050	6.606900	5.658080	5.137750	5.149390	4.777270	**4.640320**	6.353000	6.168310

TABLE 13 – Temps d'exécution (secondes) pour N=1023–1025 et 2046–2050 (OMP_NUM_THREADS=4)

Taille	16	32	64	128	256	512	1024	2048
1023	0.893265	0.710967	0.673102	0.687309	0.610476	**0.597203**	0.718978	0.721888
1024	0.878508	0.702151	0.630565	0.642346	0.554412	**0.572237**	0.722451	0.721958
1025	0.787681	0.683897	0.629180	0.620790	0.593019	**0.569430**	0.733281	0.726196
2046	6.484090	5.388030	4.771340	4.863140	4.521920	**4.354810**	6.361110	6.156460
2048	8.104590	5.852990	5.008800	4.935920	**4.397090**	4.335210	6.009580	6.201340
2050	6.468300	5.497170	4.862740	4.899280	4.626760	**4.536810**	7.814800	6.157220

TABLE 14 – Temps d'exécution (secondes) pour N=1023–1025 et 2046–2050 (OMP_NUM_THREADS=8)

Taille	16	32	64	128	256	512	1024	2048
1023	0.821436	0.677631	0.628888	0.630129	0.576028	**0.556613**	0.728713	0.734240
1024	0.883528	0.693663	0.627859	0.621725	0.562800	**0.598684**	0.725217	0.711722
1025	0.804246	0.673618	0.621317	0.648724	0.598520	**0.583687**	0.739598	0.741214
2046	6.487840	5.348190	4.750010	4.820080	4.515920	**4.173970**	6.126790	6.040050
2048	9.409870	6.329540	7.120780	5.506070	**5.147080**	5.780910	9.244030	6.967800
2050	6.795820	5.687700	5.042640	5.096350	4.772660	**4.619370**	6.553810	6.273280

TABLE 15 – Temps d'exécution (secondes) pour N=1023–1025 et 2046–2050 (OMP_NUM_THREADS=16)

Taille	16	32	64	128	256	512	1024	2048
1023	0.838795	0.696357	0.666950	0.674836	0.629240	**0.610860**	0.720595	0.720974
1024	0.918404	0.713719	0.674944	0.652920	0.604920	**0.586087**	0.719093	0.717308
1025	0.824739	0.688619	0.671568	0.685657	0.612800	**0.594540**	0.743956	0.740024
2046	6.560780	5.401080	4.887390	4.881760	4.557290	**4.376350**	6.259490	6.093770
2048	8.136170	5.905110	5.063990	5.015860	4.471350	**4.386020**	6.060130	6.269390
2050	6.379520	6.049400	4.800330	4.951530	4.595810	**4.423910**	6.323920	6.200000

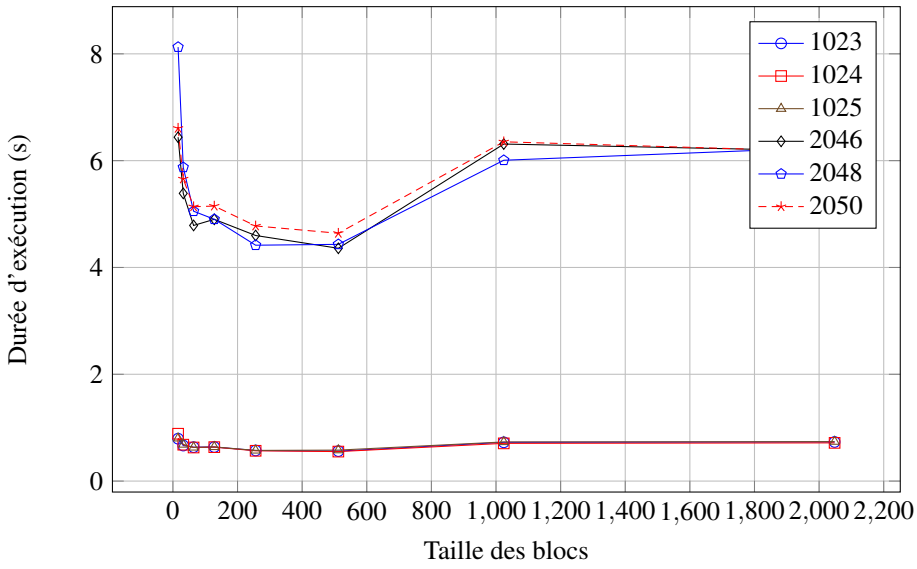

FIGURE 3 – Durées d'exécution en fonction de la taille des blocs pour chaque taille de problème. (OMP_NUM_THREADS=1)

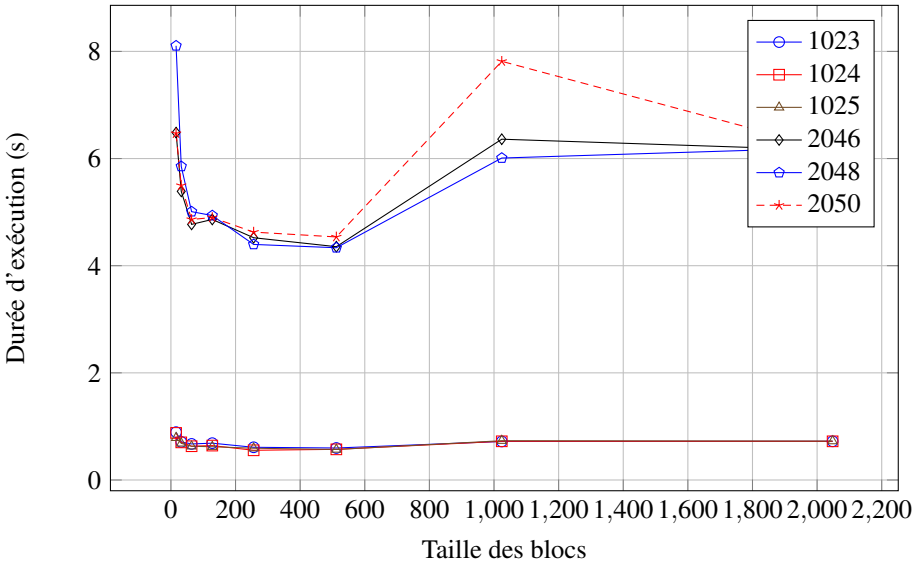

FIGURE 4 – Durées d'exécution en fonction de la taille des blocs pour chaque taille de problème. (OMP_NUM_THREADS=4)

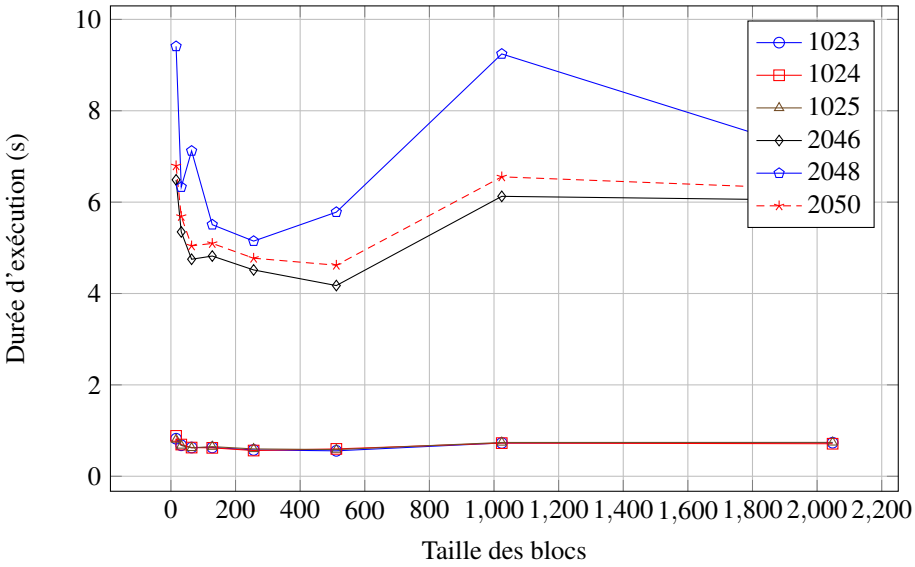

FIGURE 5 – Durées d'exécution en fonction de la taille des blocs pour chaque taille de problème. (OMP_NUM_THREADS=8)

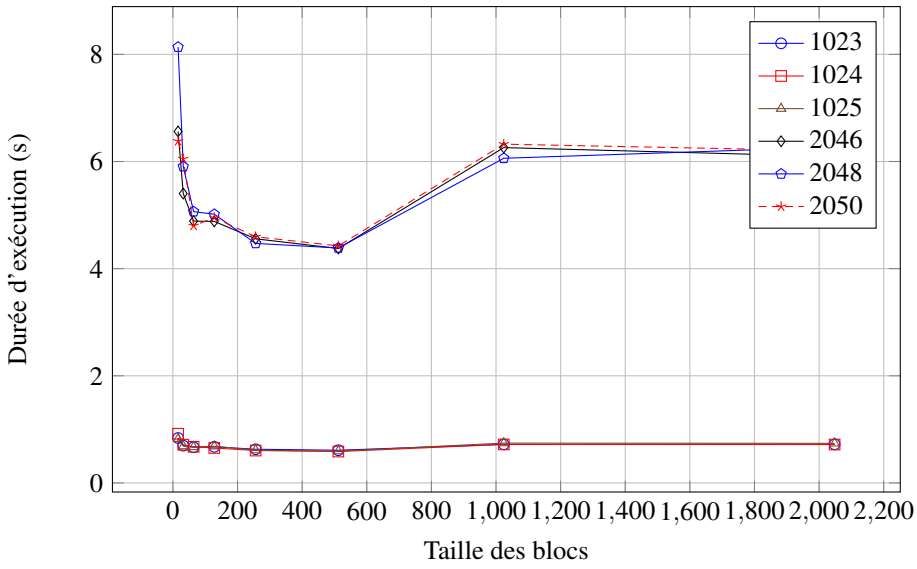

FIGURE 6 – Durées d'exécution en fonction de la taille des blocs pour chaque taille de problème. (OMP_NUM_THREADS=16)

Mesure de l'accélération Dans cette section, nous nous concentrons uniquement sur les tailles de blocs **256** et **512** car ce sont les seules qui conduisent à des temps d'exécution minimaux. Pour chacune de ces tailles, nous avons calculé la moyenne des temps CPU mesurés sur les six dimensions testées (N = 1023, 1024, 1025, 2046, 2048, 2050) pour chaque configuration de threads (OMP_NUM_THREADS = 1, 4, 8, 16). La moyenne obtenue pour le cas 1 thread sert de référence pour calculer l'accélération (speedup) selon la formule suivante :

$$\text{Speedup} = \frac{T_{\text{moyenne}}(1 \text{ thread})}{T_{\text{moyenne}}(n \text{ threads})}.$$

Taille de bloc 256 Les moyennes des temps CPU obtenues sont les suivantes :

— OMP_NUM_THREADS = 1 : $T_1(256) \approx 2.58\,\text{s}$

— OMP_NUM_THREADS = 4 : $T_4(256) \approx 2.55\,\text{s}$

— OMP_NUM_THREADS = 8 : $T_8(256) \approx 2.70\,\text{s}$

— OMP_NUM_THREADS = 16 : $T_{16}(256) \approx 2.58\,\text{s}$

Les accélérations correspondantes sont alors :

— 1 thread : Speedup = 1.00 (référence)

— 4 threads : Speedup $\approx \dfrac{2.58}{2.55} \approx 1.01$

— 8 threads : Speedup $\approx \dfrac{2.58}{2.70} \approx 0.96$

— 16 threads : Speedup $\approx \dfrac{2.58}{2.58} \approx 1.00$

Taille de bloc 512 Les moyennes des temps CPU obtenues pour la taille de bloc 512 sont :

— OMP_NUM_THREADS = 1 : $T_1(512) \approx 2.52\,\text{s}$

— OMP_NUM_THREADS = 4 : $T_4(512) \approx 2.49\,\text{s}$

— OMP_NUM_THREADS = 8 : $T_8(512) \approx 2.72\,\text{s}$

— OMP_NUM_THREADS = 16 : $T_{16}(512) \approx 2.50\,\text{s}$

Les accélérations correspondantes sont :

— 1 thread : Speedup = 1.00 (référence)

— 4 threads : Speedup $\approx \dfrac{2.52}{2.49} \approx 1.01$

— 8 threads : Speedup $\approx \dfrac{2.52}{2.72} \approx 0.93$

— 16 threads : Speedup $\approx \dfrac{2.52}{2.50} \approx 1.01$

TABLE 16 – Accélérations moyennes (baseline : OMP_NUM_THREADS = 1) pour les tailles de blocs 256 et 512

Taille de bloc	1 thread	4 threads	8 threads	16 threads
256	1.00	1.01	0.96	1.00
512	1.00	1.01	0.93	1.01

Tableau récapitulatif des accélérations pour les tailles de blocs 256 et 512 Pour une taille de bloc fixée, nous représentons l'évolution de l'accélération (speedup) en fonction du nombre de threads.

Évolution des accélérations pour la taille de bloc 256

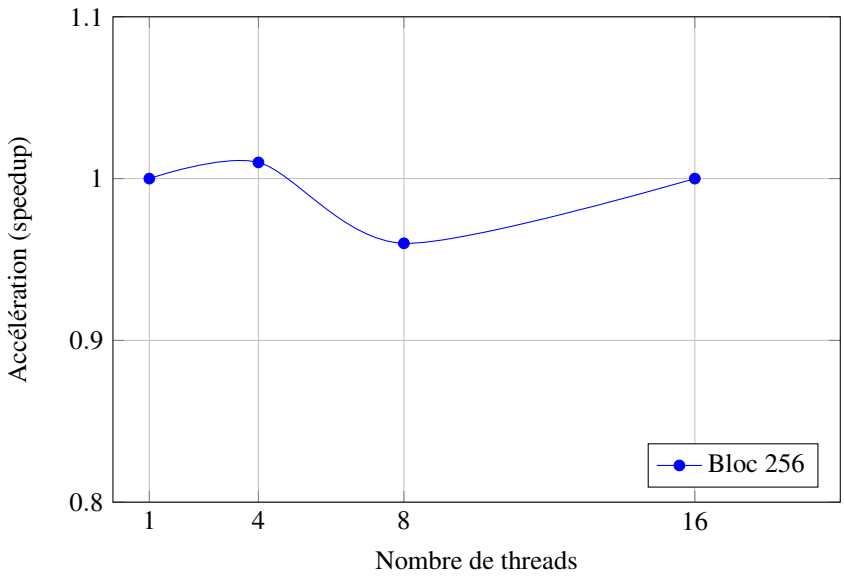

FIGURE 7 – Évolution du speedup en fonction du nombre de threads pour une taille de bloc de 256.

Évolution des accélérations pour la taille de bloc 512 Remarques :

— Pour la taille de bloc 256, une légère amélioration est observée avec 4 threads (speedup ≈ 1.01) et 16 threads (speedup ≈ 1.00), tandis que 8 threads présente une performance légèrement inférieure (speedup ≈ 0.96).

— Pour la taille de bloc 512, on note également une amélioration avec 4 et 16 threads (speedup ≈ 1.01), et une dégradation avec 8 threads (speedup ≈ 0.93).

Commentaire :

— Avec 4 threads : amélioration légère (speedup ≈ 1.01)

Chaque thread peut s'exécuter sur un cœur distinct sans entrer en conflit avec un autre thread.

— Avec 8 threads : légère dégradation (speedup ≈ 0.96 pour 256, ≈ 0.93 pour 512)

L'Hyper-Threading crée une compétition entre les threads d'un même cœur, ce qui peut augmenter les latences d'accès au cache. L'augmentation du nombre de threads entraîne une saturation des unités d'exécution et des ressources mémoire.

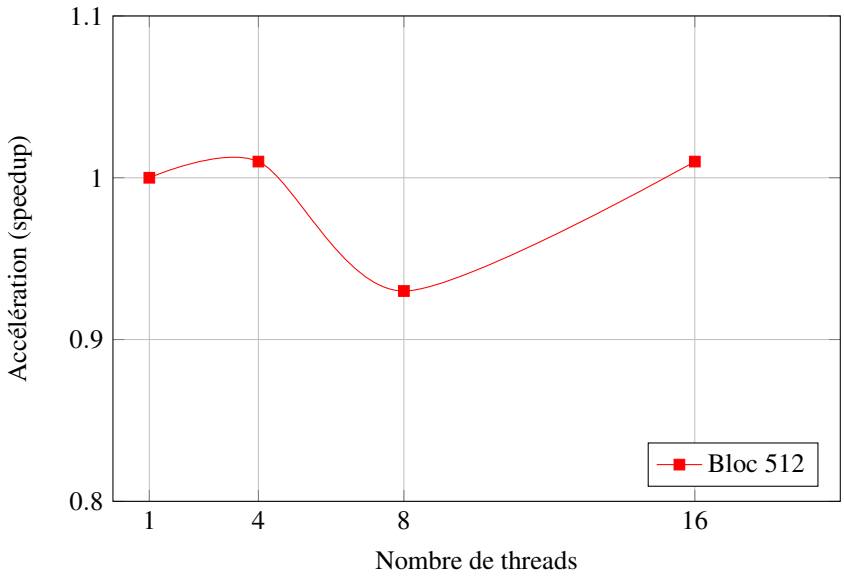

FIGURE 8 – Évolution du speedup en fonction du nombre de threads pour une taille de bloc de 512.

Comparaison avec le produit matrice-matrice "scalaire"

1. **Observation** : L'accélération parallèle est inférieure pour la version par blocs.
2. **Explication** : La version parallèle du produit par blocs peut introduire des surcoûts liés à la synchronisation entre threads et à la gestion de la répartition des blocs. Chaque thread peut devoir attendre la fin des calculs sur certains blocs pour pouvoir recombiner les résultats, ce qui limite l'efficacité du parallélisme comparé à un produit matriciel simple.

1.8 Comparaison avec BLAS

Nous avons installé libmkl-dev pour pouvoir effectuer les tests. Le code du fichier test_product_matrice_blas.cpp appelle explicitement la fonction BLAS dgemm_ via l'instruction :

```
extern "C" void dgemm_(char const& trA, char const& trB, int const& m, int const& n,
int const& k,
double const& alpha, double const* A, int const& ldA, double const* B,
int const& ldB, double const& beta, double* C, int const& ldC );
```

Pour la compilation, nous avons utilisé la commande suivante :

```
g++ -O3 -m64 -I${MKLROOT}/include test_product_matrice_blas.cpp Matrix.cpp \
    -L${MKLROOT}/lib/intel64 -lmkl_intel_lp64 -lmkl_core -lmkl_intel_thread -liomp5 -
    lpthread \
    -o test_product_matrice_blas
./test_product_matrice_blas
```

Explication des options

— `-O3` : Active un niveau d'optimisation agressif pour améliorer les performances du code. Sans cette option, les performances sont plus variables sur différents tests (0.0262079, 0.0412135, 0.0267867, 0.0324142 secondes) contre 0.0277023, 0.0289546, 0.0281378 secondes avec cette option. Le test étant effectué avec une dimension de 1024.

— `-m64` : Indique une compilation en mode 64 bits.

— `-I${MKLROOT}/include` : Spécifie le chemin vers les fichiers d'en-tête de la bibliothèque Intel MKL.

— `test_product_matrice_blas.cpp Matrix.cpp` : Liste des fichiers sources à compiler.

— `-L${MKLROOT}/lib/intel64` : Spécifie le répertoire contenant les bibliothèques MKL à linker.

— `-lmkl_intel_lp64` : Lie la version LP64 (long pointer 64 bits) de MKL pour une compatibilité avec les types de données 64 bits.

— `-lmkl_core` : Lie le cœur de la bibliothèque MKL, qui contient les implémentations des algorithmes de calcul.

— `-lmkl_intel_thread` : Utilise la version multi-thread de MKL pour améliorer les performances sur plusieurs cœurs.

— `-liomp5` : Lie la bibliothèque OpenMP d'Intel pour la gestion du parallélisme.

— `-lpthread` : Lie la bibliothèque POSIX Threads, nécessaire pour le support multi-threading.

— `-o test_product_matrice_blas` : Spécifie le nom de l'exécutable généré.

Les **performances obtenues avec BLAS sont meilleures** que celles obtenues avec notre produit matrice matrice par blocs de 512 et parallélisé (4 threads). Le rapport moyen des performances $\approx \frac{T_4(512)}{0.085} = \frac{2.49}{0.085} = 29.29$.

Dimension	Temps CPU (s)	MFlops
1023	0.0251497	85138.2
1024	0.0251589	85356.7
1025	0.0264412	81455.6
2046	0.143318	119522
2048	0.152871	112381
2050	0.140413	122711
Moyenne	0.0852253	102094.25

TABLE 17 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices obtenus avec Blas.

2 Parallélisation MPI

2.1 Circulation d'un jeton dans un anneau.

Code :

```
1
2       #include <stdio.h>
3       #include <mpi.h>
4
5       int main(int argc, char *argv[]) {
6           int rank, nbp, token;
7
8           /* Initialisation de l'environnement MPI */
9           MPI_Init(&argc, &argv);
10          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11          MPI_Comm_size(MPI_COMM_WORLD, &nbp);
12
13          /* On verifie qu'il y a au moins 2 processus */
14          if(nbp < 2) {
15              if(rank == 0)
16                  fprintf(stderr, "Ce programme necessite au moins 2 processus.\n");
17              MPI_Finalize();
18              return 1;
19          }
20
21          /* Processus de rang 0 initialise le jeton et l'envoie au processus de rang 1
     */
22          if (rank == 0) {
23              token = 1;  // initialisation du jeton
24              printf("Processus %d envoie le jeton %d vers le processus 1\n", rank,
    token);
25              MPI_Send(&token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
26
27              /* Processus 0 attend ensuite de recevoir le jeton venant du dernier
    processus */
28              MPI_Recv(&token, 1, MPI_INT, nbp - 1, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
```

```
                printf("Processus %d a recu le jeton %d depuis le processus %d\n", rank, token, nbp - 1);
            }
            else {
                /* Tous les autres processus recoivent le jeton du processus precedent */
                MPI_Recv(&token, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                token++;  // incrementation du jeton
                printf("Processus %d a recu le jeton et l'incremente a %d\n", rank, token);

                /* Envoi du jeton au processus suivant.
                   Le processus nbp-1 envoie vers le processus 0 pour fermer l'anneau. */
                int dest = (rank + 1) % nbp;
                MPI_Send(&token, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
            }

            MPI_Finalize();
            return 0;
        }
```

Listing 1 – Code source de `jeton.c`

Exécution

1. mpicc jeton.c -o jeton
2. mpirun -np 4 ./jeton

4 étant la valeur de nbp.

Résultat

— Processus 0 envoie le jeton 1 vers le processus 1

— Processus 1 a reçu le jeton et l'incrémente à 2

— Processus 2 a reçu le jeton et l'incrémente à 3

— Processus 3 a reçu le jeton et l'incrémente à 4

— Processus 0 a reçu le jeton 4 depuis le processus 3

Calcul très approché de π

Version séquentielle en c .

```
        /* pi_seq.c */
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>
        #include <math.h>

        double approximate_pi(unsigned long nbSamples) {
            unsigned long nbDarts = 0;
            for (unsigned long i = 0; i < nbSamples; i++) {
                /* Generer un point dans [-1, 1] */
                double x = (double)rand() / RAND_MAX * 2.0 - 1.0;
                double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
                if (x*x + y*y <= 1.0)
```

```
                nbDarts++;
            }
            double ratio = (double)nbDarts / nbSamples;
            return 4.0 * ratio;
        }

        int main(void) {
            unsigned long nbSamples = 10000000UL;
            srand(time(NULL));
            clock_t tdeb = clock();
            double pi = approximate_pi(nbSamples);
            clock_t tfin = clock();
            double temps = (double)(tfin - tdeb) / CLOCKS_PER_SEC;
            printf("Approximation de pi (sequentiel) = %f\n", pi);
            printf("Temps d'execution = %f secondes\n", temps);
            return 0;
        }
```

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

```
        /* pi_openmp.c */
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>
        #include <math.h>
        #include <omp.h>

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

            #pragma omp parallel
            {
                unsigned int seed = time(NULL) ^ omp_get_thread_num();
                unsigned long localCount = 0;
                #pragma omp for
                for (unsigned long i = 0; i < nbSamples; i++) {
                    double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
                    double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
                    if (x*x + y*y <= 1.0)
                        localCount++;
                }
                #pragma omp atomic
                nbDarts += localCount;
            }
            double ratio = (double)nbDarts / nbSamples;
            return 4.0 * ratio;
        }

        int main(void) {
            unsigned long nbSamples = 10000000UL;
            double tdeb = omp_get_wtime();
            double pi = approximate_pi(nbSamples);
            double tfin = omp_get_wtime();
            printf("Approximation de pi (OpenMP) = %f\n", pi);
            printf("Temps d'execution = %f secondes\n", tfin - tdeb);
```

```
            return 0;
        }
```

Nombre de threads	Approximation de π (OpenMP)	Temps d'exécution (secondes)	Accélération
1 thread	3.141976	0.118586	1
4 threads	3.142339	0.055988	2,118061013
8 threads	3.140559	0.054829	2,162833537
16 threads	3.141930	0.04843	2,448606236

TABLE 18 – Résultats de l'approximation de π avec OpenMP pour différents nombres de threads.

Mesure de l'accélération (fig 18)

Version en mémoire distribuée avec MPI en C .

```c
        /* pi_mpi.c */
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>
        #include <math.h>
        #include <mpi.h>

        int main(int argc, char *argv[]) {
            int rank, nbp;
            MPI_Init(&argc, &argv);
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
            MPI_Comm_size(MPI_COMM_WORLD, &nbp);

            unsigned long nbSamplesTotal = 10000000UL;
            /* Chaque processus traite une portion des echantillons */
            unsigned long nbSamples = nbSamplesTotal / nbp;
            unsigned long nbDarts_local = 0;

            /* Utilisation d'une graine differente pour chaque processus */
            unsigned int seed = time(NULL) ^ rank;

            double tdeb = MPI_Wtime();
            for (unsigned long i = 0; i < nbSamples; i++) {
                double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
                double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
                if (x*x + y*y <= 1.0)
                    nbDarts_local++;
            }

            unsigned long nbDarts_total = 0;
            MPI_Reduce(&nbDarts_local, &nbDarts_total, 1, MPI_UNSIGNED_LONG, MPI_SUM, 0,
    MPI_COMM_WORLD);
            double tfin = MPI_Wtime();

            if (rank == 0) {
                double ratio = (double)nbDarts_total / (nbSamples * nbp);
                double pi = 4.0 * ratio;
```

```
                printf("Approximation de pi (MPI) = %f\n", pi);
                printf("Temps d'execution (MPI) = %f secondes\n", tfin - tdeb);
            }

            MPI_Finalize();
            return 0;
        }
```

Nombre de processus	Approximation de π (MPI)	Temps d'exécution (secondes)	Accélération
1	3.141547	0.168330	1
2	3.141825	0.084710	1,98713257
4	3.142389	0.072163	2,332635838
6	*Erreur*	*Non disponible*	Non disponible
8	*Erreur*	*Non disponible*	Non disponible

TABLE 19 – Résultats de l'approximation de π avec MPI pour différents nombres de processus.

Mesure de l'accélération (fig 19)

Commentaire Causes des erreurs :

— Nombre de slots disponibles insuffisant

— Open MPI utilise le concept de slots pour allouer des processus. Un slot correspond à une unité d'allocation (généralement un cœur ou un thread).

— Notre processeur (Intel Core i5-10210U) a 4 cœurs physiques et 8 threads logiques (grâce à la technologie Hyper-Threading (fig 1)).

— Par défaut, Open MPI limite le nombre de slots au nombre de cœurs physiques ou threads disponibles. Si on essaie de lancer plus de processus que de slots disponibles, l'erreur se produit.

Version MPI en python avec mpi4py .

```
#!/usr/bin/env python3
# pi_mpi.py
from mpi4py import MPI
import time, numpy as np

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

nb_samples_total = 40_000_000
nb_samples = nb_samples_total // size  # repartition uniforme des echantillons

# Debut de la mesure de temps (commence avant le calcul)
start = MPI.Wtime()

# Chaque processus initialise sa graine pour generer des nombres aleatoires differents
np.random.seed(int(time.time()) ^ rank)

```

```
        # Generation des points aleatoires dans [-1, 1] pour x et y
        x = np.random.uniform(-1, 1, nb_samples)
        y = np.random.uniform(-1, 1, nb_samples)

        # Compter le nombre de points tombant dans le cercle unite
        local_count = np.count_nonzero(x*x + y*y <= 1.0)

        # Reduire les resultats de tous les processus (somme)
        total_count = comm.reduce(local_count, op=MPI.SUM, root=0)

        # Fin de la mesure de temps
        end = MPI.Wtime()

        # Le processus maitre calcule l'approximation de pi et affiche le temps d'
    execution
        if rank == 0:
            approx_pi = 4.0 * total_count / nb_samples_total
            print(f"Approximation de pi (mpi4py) = {approx_pi}")
            print(f"Temps d'execution = {end - start} secondes")
```

Listing 2 – Code source de `pi_mpi.py`

Nombre de processus	Approximation de π (MPI)	Temps d'exécution (secondes)	Accélération
1	3.141979	1.43688583	1
2	3.1421147	0.709724572	2,024568243
4	3.1417664	0.684759401	2,098380581
6	*Erreur*	*Non disponible*	Non disponible
8	*Erreur*	*Non disponible*	Non disponible

TABLE 20 – Résultats de l'approximation de π avec MPI pour différents nombres de processus.

2.2 Diffusion d'un entier dans un réseau hypercube.

Code :

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <mpi.h>

        int main(int argc, char* argv[]) {
            int rank, nbp, d;
            MPI_Init(&argc, &argv);
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);
            MPI_Comm_size(MPI_COMM_WORLD, &nbp);

            // Determination de la dimension d :
            // Si un argument est fourni, on l'utilise ; sinon, on deduit d a partir du
    nb de processus.
            if (argc > 1) {
                d = atoi(argv[1]);
            } else {
                // Deduire d si nbp est une puissance de 2.
                d = 0;
```

```
18                int tmp = nbp;
19                while (tmp > 1) {
20                    if (tmp % 2 != 0) {
21                        if (rank == 0)
22                            fprintf(stderr, "Erreur : Le nombre de processus doit etre
    une puissance de 2.\n");
23                        MPI_Finalize();
24                        exit(EXIT_FAILURE);
25                    }
26                    tmp /= 2;
27                    d++;
28                }
29            }
30
31            // Verification : le nombre de processus doit etre 2^d.
32            if (nbp != (1 << d)) {
33                if (rank == 0)
34                    fprintf(stderr, "Erreur : Pour un hypercube de dimension %d, il faut
    2^d processus (ici %d processus).\n", d, nbp);
35                MPI_Finalize();
36                exit(EXIT_FAILURE);
37            }
38
39            int token;  // le jeton a diffuser
40            double tdeb, tfin;
41
42            // Mesurer le temps de diffusion
43            tdeb = MPI_Wtime();
44
45            // 1. Cas de l'hypercube de dimension 1 (2 processus) :
46            //    - Si d vaut 1, alors seul 2 processus sont utilises.
47            //    - Le processus 0 initialise et envoie au processus 1.
48            // Ce meme algorithme fonctionne pour d = 2, d = 3 et d = d general.
49            if (rank == 0) {
50                token = 12345;  // valeur choisie par le programmeur
51            }
52
53            // Diffusion dans un hypercube en d etapes.
54            // Pour chaque etape i, chaque processus echange avec son partenaire defini
    par rank xor (1 << i).
55            for (int i = 0; i < d; i++) {
56                int partner = rank ^ (1 << i);
57                if (rank < partner) {
58                    // Le processus avec un rang plus faible envoie d'abord
59                    MPI_Send(&token, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
60                } else {
61                    // Celui avec le rang plus eleve recoit
62                    MPI_Recv(&token, 1, MPI_INT, partner, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
63                }
64            }
65
66            tfin = MPI_Wtime();
67
68            // Affichage : chaque processus affiche la valeur recue et le temps de
    diffusion.
```

```
69              printf("Processus %d : token = %d, temps de diffusion = %f secondes\n", rank,
    token, tfin - tdeb);
70
71              MPI_Finalize();
72              return 0;
73          }
```

Listing 3 – Code source de `hypercube.c`

mpirun -np 4 ./compute_hypercube 2

- Processus 0 : token = 12345, temps de diffusion = 0.000027 secondes
- Processus 1 : token = 12345, temps de diffusion = 0.000085 secondes
- Processus 2 : token = 12345, temps de diffusion = 0.000027 secondes
- Processus 3 : token = 12345, temps de diffusion = 0.000116 secondes