2024/25

Nom : DADA SIMEU CÉDRIC DAREL

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

Titre : Compte rendu TP 1

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 :

```
1       namespace {
2           void prodSubBlocks(int iRowBlkA, int iColBlkB, int iColBlkA, int szBlock,
3                              const Matrix& A, const Matrix& B, Matrix& C) {
4               for (int j = iColBlkB; j < std::min(B.nbCols, iColBlkB + szBlock); ++j) {
5                   for (int k = iColBlkA; k < std::min(A.nbCols, iColBlkA + szBlock); ++
    k) {
6                       const double b_kj = B(k, j);
7                       for (int i = iRowBlkA; i < std::min(A.nbRows, iRowBlkA + szBlock)
    ; ++i) {
8                           C(i, j) += A(i, k) * b_kj;
9                       }
10                  }
11              }
12          }
13
```

```
14          int findOptimalBlockSize(const Matrix& A, const Matrix& B) {
15              std::vector<int> sizes = {16, 32, 64, 128, 256};
16              double best_time = 1e9;
17              int best_size = 64;
18
19              for (int sz : sizes) {
20                  Matrix C(A.nbRows, B.nbCols, 0.0);
21                  double start = omp_get_wtime();
22
23                  for (int I = 0; I < A.nbRows; I += sz) {
24                      for (int J = 0; J < B.nbCols; J += sz) {
25                          for (int K = 0; K < A.nbCols; K += sz) {
26                              prodSubBlocks(I, J, K, sz, A, B, C);
27                          }
28                      }
29                  }
30
31                  double duration = omp_get_wtime() - start;
32                  if (duration < best_time) {
33                      best_time = duration;
34                      best_size = sz;
35                  }
36              }
37
38              std::cout << "[OPTIM] Taille de bloc optimale: " << best_size << std::
    endl;
39              return best_size;
40          }
41          } // namespace
42
43          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) {       // Parcours lignes A
48                  for (int J = 0; J < B.nbCols; J += szBlock) {   // Parcours colonnes
    B
49                      for (int K = 0; K < A.nbCols; K += szBlock) { // Parcours commun
50                          prodSubBlocks(I, J, K, szBlock, A, B, C);
51                      }
52                  }
53              }
54              return C;
55          }
```

Conclusion : Nous voyons que globalement, la taille optimale est **256**.

Commentaire : Notre machine (avec un Intel(R) Core(TM) i5-10210U) possède 6MiB de cache L3 et 1MiB répartis sur 4 cœurs en cache L2, soit environ 256KiB par cœur (voir fig **??** et **??**), et il se trouve que la taille 256 offre le meilleur compromis en termes de :

— Réutilisation des données : Un bloc de taille 256×256 en double précision (où chaque double occupe 8 octets) représente environ 256×256×8 = 524 288 octets (soit 512 KiB). Ce bloc (ou la combinaison de blocs issus de A, B et C) est dimensionné de façon à exploiter au mieux la capacité des caches de niveau L2 et L3.

— Amortissement du coût de chargement : Avec une taille de bloc bien choisie, une fois les sous-matrices chargées dans le cache, de nombreux calculs peuvent être effectués sans accéder à la mémoire principale.

— Optimisation pour la hiérarchie mémoire de votre machine : Les tailles de caches (L1, L2, L3) et la structure de la mémoire influencent fortement le choix d'un bloc optimal. Le fait que 256 apparaisse comme la meilleure taille dans vos tests expérimentaux est lié aux capacités spécifiques de votre ordinateur.

```
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

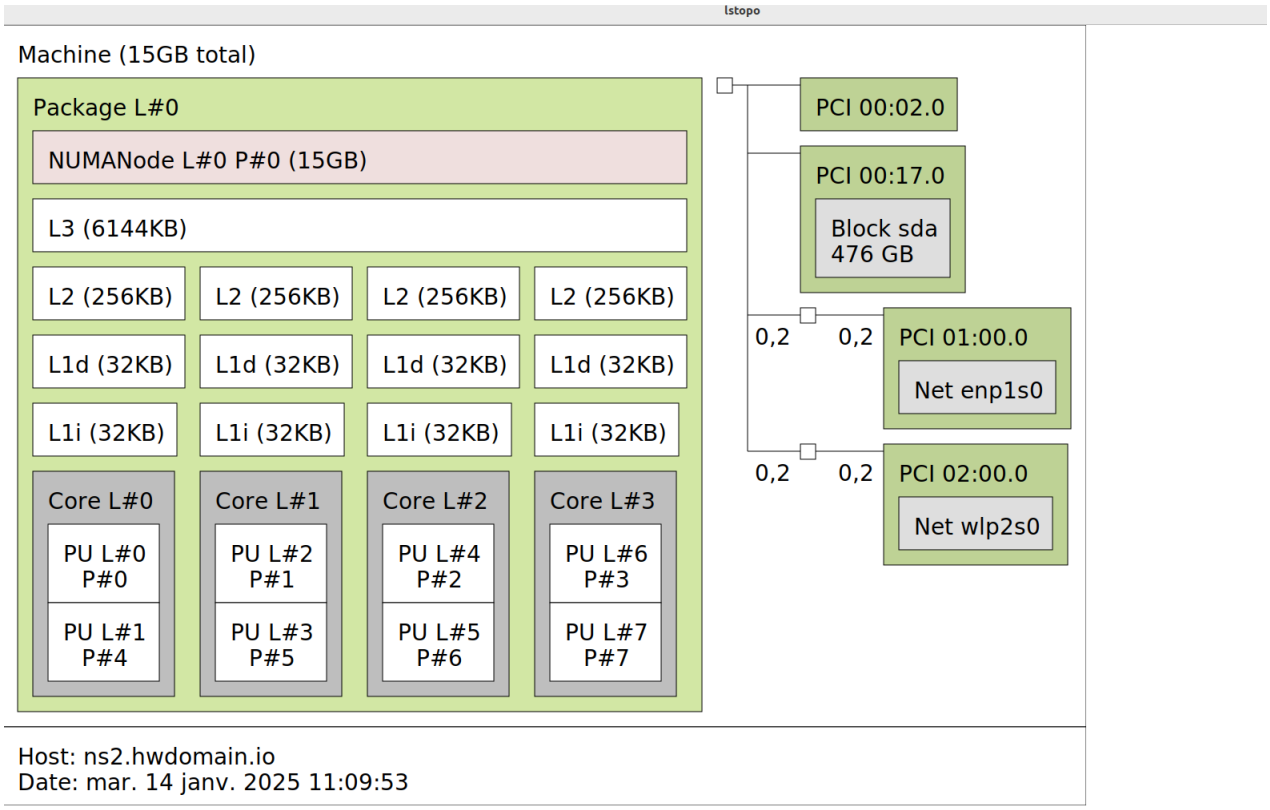

FIGURE 2 – Résultat de la commande lstopo : Nous pouvons visualiser les tailles des caches

TABLE 11 – Temps d'exécution (secondes) pour N=1023-1025

Taille	16	32	64	128	256
1023	0.769	0.675	0.638	0.631	**0.582**
1024	1.199	**1.017**	1.438	1.455	1.218
1025	1.549	1.180	0.902	0.766	**0.566**
2046	6.25	7.67	5.15	6.19	**4.79**
2048	11.07	9.38	7.55	7.45	**7.30**
2050	8.69	6.86	6.42	6.19	**4.94**

TABLE 12 – OMP_NUM_THREADS=1

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

La version par bloc devrait être significativement plus rapide grâce à une meilleure utilisation des caches. Pour une taille de blocs de **256**, et un thread, nous avons une durée de moyenne (pour les dimensions 1023, 1024 et 1025) de **0,789 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;
}
```

Dimension	Temps CPU (s)	MFlops
1023	0.625676	3422.21
1024	1.33775	1605.29
1025	1.24341	1732.16
Moyenne	1,06	2253,22

TABLE 13 – Performances pour OMP_NUM_THREADS=1.

.

.

.

Dimension	Temps CPU (s)	MFlops
1023	0.224386	9542.48
1024	0.23782	9029.87
1025	0.403212	5341.56
Moyenne	0,288	7971,30

TABLE 14 – Performances pour OMP_NUM_THREADS=4.

Dimension	Temps CPU (s)	MFlops
1023	0.255513	8380.01
1024	0.257188	8349.86
1025	0.228108	9441.95
Moyenne	0,247	8723,94

TABLE 15 – Performances pour OMP_NUM_THREADS=16.

Mesure de l'accélération Nous obtenons une accélération de $\frac{1.06}{0.288} = 3,68$ pour 4 threads et de $\frac{1.06}{0.247} = 4,29$ pour 16 threads.

Comparaison avec le produit matrice-matrice "scalaire"

1. **Observation** : L'accélération parallèle est supérieure pour la version par blocs.
2. **Explications** :
   - Réduction des conflits de cache : Les blocs isolent les zones mémoire travaillées par chaque thread → pas de false sharing.
   - Localité mémoire préservée : Chaque thread travaille sur des sous-blocs contigus → meilleure utilisation du cache L1/L2.
   - Granularité adaptée : La taille des blocs (ex : 256) équilibre charge de travail et utilisation du cache.

Raison : Le surcoût de synchronisation est amorti par le volume de calcul dans chaque bloc.

1.8 Comparaison avec BLAS

Dimension	Temps CPU (s)	MFlops
1023	0.0473309	45238.9
1024	0.0609969	35206.4
1025	0.057084	37730
Moyenne	0.055	39391,77

TABLE 16 – Temps de calcul et performances en MFlops pour différentes dimensions de matrices obtenus avec Blas.

Les performances obtenues avec BLAS sont meilleures que celles obtenues avec notre produit matrice matrice par blocs parallélisé (16 threads). Le rapport moyen des performances $\approx \frac{0,247}{0.055} = 4.49$.

La meilleure version est celle de BLAS.

2 Parallélisation MPI

2.1 Circulation d'un jeton dans un anneau.

Code :

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

int main(int argc, char *argv[]) {
    int rank, nbp, token;

    /* Initialisation de l'environnement MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nbp);

    /* On verifie qu'il y a au moins 2 processus */
    if(nbp < 2) {
        if(rank == 0)
            fprintf(stderr, "Ce programme necessite au moins 2 processus.\n");
        MPI_Finalize();
        return 1;
    }

    /* Processus de rang 0 initialise le jeton et l'envoie au processus de rang 1 */
    if (rank == 0) {
        token = 1;  // initialisation du jeton
        printf("Processus %d envoie le jeton %d vers le processus 1\n", rank, token);
        MPI_Send(&token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

        /* Processus 0 attend ensuite de recevoir le jeton venant du dernier processus */
        MPI_Recv(&token, 1, MPI_INT, nbp - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Processus %d a re u le jeton %d depuis le processus %d\n", rank, token, nbp - 1);
    }
    else {
        /* Tous les autres processus re oivent 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 re u le jeton et l'incremente    %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);
    }
```

```
42
43              MPI_Finalize();
44              return 0;
45          }
```

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 .

```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; // par exemple 10 millions de points
    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 .

```c
/* 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 17 – Résultats de l'approximation de π avec OpenMP pour différents nombres de threads.

Mesure de l'accélération (fig ??)

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;

    /* Utiliser 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 18 – Résultats de l'approximation de π avec MPI pour différents nombres de processus.

Mesure de l'accélération (fig ??)

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 **??**)).

— 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 g n rer des nombres aleatoires diff rents
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)

# R duire 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")
```

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

2.2 Diffusion d'un entier dans un réseau hypercube.

Code :

```c
#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;
        int tmp = nbp;
        while (tmp > 1) {
            if (tmp % 2 != 0) {
                if (rank == 0)
                    fprintf(stderr, "Erreur : Le nombre de processus doit etre une puissance de 2.\n");
                MPI_Finalize();
                exit(EXIT_FAILURE);
            }
            tmp /= 2;
            d++;
        }
    }

    // Verification : le nombre de processus doit etre 2^d.
    if (nbp != (1 << d)) {
        if (rank == 0)
            fprintf(stderr, "Erreur : Pour un hypercube de dimension %d, il faut 2^d processus (ici %d processus).\n", d, nbp);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

```

```
            int token;  // le jeton a diffuser
            double tdeb, tfin;

            // Mesurer le temps de diffusion
            tdeb = MPI_Wtime();

            // 1. Cas de l'hypercube de dimension 1 (2 processus) :
            //    - Si d vaut 1, alors seul 2 processus sont utilises.
            //    - Le processus 0 initialise et envoie au processus 1.
            // Ce meme algorithme fonctionne pour d = 2, d = 3 et d = d general.
            if (rank == 0) {
                token = 12345;  // valeur choisie par le programmeur
            }

            // Diffusion dans un hypercube en d etapes.
            // Pour chaque etape i, chaque processus echange avec son partenaire defini par rank xor (1 << i).
            for (int i = 0; i < d; i++) {
                int partner = rank ^ (1 << i);
                if (rank < partner) {
                    // Le processus avec un rang plus faible envoie d'abord
                    MPI_Send(&token, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
                } else {
                    // Celui avec le rang plus eleve recoit
                    MPI_Recv(&token, 1, MPI_INT, partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                }
            }

            tfin = MPI_Wtime();

            // Affichage : chaque processus affiche la valeur recue et le temps de diffusion.
            printf("Processus %d : token = %d, temps de diffusion = %f secondes\n", rank, token, tfin - tdeb);

            MPI_Finalize();
            return 0;
        }
```

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