OS202 Travaux dirigés n°1

4 février 2025

Xingzi ZHANG
2024-2025

Table des matières

1. Produit matrice–matrice

1.1. Mesure du temps de calcul

Pour mesurer le temps de calcul, nous avons exécuté le programme sur des matrices de dimensions 1023, 1024 et 1025. Les résultats sont les suivants :

Matrice Dimension	Temps (s)	MFlops	Temps Moyen (s)	MFlops Moyen
1023	8.44484, 7.82472, 7.83068	253.551, 273.645, 273.437	8.03308	266.211
1024	11.044, 10.723, 10.9328	194.448, 200.268, 196.426	10.89993	197.714
1025	7.85553, 7.82043, 8.01118	274.174, 275.404, 268.847	7.89505	272.808

Table 1.1 : Temps de calcul et MFlops pour différentes dimensions.

Analyse :

- Les performances pour les dimensions 1023 et 1025 sont similaires et supérieures à celles pour 1024.
- La performance inférieure pour 1024 est attribuée à des effets de cache et à l'alignement mémoire.

1.2. Première optimisation : Permutation des boucles

Pour optimiser le produit matrice-matrice, nous avons expérimenté avec plusieurs ordres d'exécution des boucles `i, j, k`, et analysé les performances en termes de temps d'exécution et de MFlops pour différentes dimensions de matrices (1023, 1024, 1025). Les résultats sont présentés dans le tableau suivant :

Ordre	1023 t(s)	1023 MFlops	1024 t(s)	1024 MFlops	1025 t(s)	1025 MFlops
`i -> j -> k`	8.03308	266.21	10.89993	197.71	7.89505	272.81
`i -> k -> j`	1.09922	1946.11	5.38586	398.98	1.12131	1922.74
`j -> i -> k`	2.07064	1035.98	5.26093	409.77	2.28154	946.06
`j -> k -> i`	0.6334	3380.83	0.62623	3429.31	0.63141	3411.83
`k -> i -> j`	7.22160	296.64	10.33923	207.07	7.10521	303.14
`k -> j -> i`	0.89771	2385.18	0.90751	2366.85	0.94777	2274.74

Table 1.2 : Temps d'exécution et MFlops pour différents ordres de boucles

Analyse des résultats :

- L'ordre `k -> j -> i` montre des performances supérieures à `i -> k -> j` et `k -> i -> j`. Cela est dû à une meilleure localité des données en mémoire cache, car les éléments de la matrice sont accédés de manière séquentielle.
- L'ordre `j -> k -> i` présente la meilleure performance globale en termes de temps d'exécution et de MFlops. Cela s'explique par le fait que cet ordre minimise les accès redondants à la mémoire et maximise l'utilisation des caches.
- L'ordre `k -> j -> i`, bien qu'efficace, est légèrement moins performant que `j -> k -> i` car il nécessite plus de transferts de données entre les caches et la mémoire principale.

Conclusion :
L'ordre optimal pour le produit matrice-matrice est `j -> k -> i`. Cet ordre permet d'exploiter pleinement la hiérarchie des mémoires (cache et RAM) en réduisant les conflits de cache et en maximisant la localité des

données. Cette conclusion est cohérente avec les principes enseignés dans le cours, qui recommandent d'ordonner les boucles de manière à minimiser les coûts d'accès mémoire.

1.3. Première parallélisation avec OpenMP

Pour paralléliser le produit matrice-matrice, nous avons utilisé OpenMP en insérant la directive `#pragma omp parallel for` dans la boucle principale du calcul matriciel. Nous avons ensuite mesuré le temps d'exécution pour différents nombres de threads, grâce à `OMP_NUM_THREADS`.

Nombre de Threads	Temps (s)	MFlops	Accélération
1	11.5555	185.841	1.000
2	11.6390	184.508	0.993
4	10.8939	197.128	1.061
8	11.8357	181.442	0.976
16	10.8939	197.128	1.061

TABLE 1.3 : Temps d'exécution et accélération en fonction du nombre de threads

L'analyse des résultats montre que l'accélération attendue n'est pas linéaire. En effet, au lieu d'une amélioration constante des performances, nous constatons que l'utilisation de 8 et 16 threads ne permet pas de réduire significativement le temps d'exécution. Cela s'explique par plusieurs facteurs, notamment la surcharge de synchronisation entre les threads et les conflits de cache.

1.4. Possibilités d'amélioration

L'accélération obtenue étant relativement faible, il est possible d'améliorer les performances du calcul parallélise en OpenMP par plusieurs méthodes :

- **Amélioration de la répartition des tâches** : L'utilisation de `schedule(dynamic, chunk_size)` pourrait aider à équilibrer la charge de travail entre les threads et minimiser les temps d'attente.
- **Optimisation des accès mémoire** : L'ajout de `collapse(2)` permettrait de paralléliser les boucles imbriquées et d'améliorer l'efficacité des caches CPU.
- **Optimisation des blocs de calcul** : Adapter la taille des blocs dans l'algorithme pourrait réduire les conflits de cache et améliorer la localité des données

1.5. Deuxième optimisation : Produit par bloc

szBlock	**MFlops (n=1024)**	**MFlops (n=2048)**	**MFlops (n=512)**
32	192.003	97.2808	535.645
64	191.036	94.6337	537.015
128	121.112	83.6708	534.811
256	159.394	62.5811	530.070
512	189.001	65.3316	529.560
1024	181.241	99.2137	512.223

TABLE 1.4 : Performance (MFLOPS) en fonction de la taille de bloc et de la dimension de la matrice.

1.6. Comparaison avec produit scalaire

Pour `n=512` : La meilleure performance est obtenue avec `szBlock=64`, atteignant **537.015 MFlops**. Ceci est dû à une bonne optimisation de l'utilisation du cache et un équilibrage entre la charge de calcul et les accès mémoire.

Pour `n=1024` : Le choix optimal est `szBlock=32`, avec une performance de **192.003 MFlops**. Les petits blocs permettent une meilleure gestion des données en cache, réduisant ainsi les conflits.

Pour `n=2048` : Pour les grandes matrices, le bloc de taille `szBlock=1024` est le plus performant avec **99.2137 MFlops**. L'augmentation de la taille des blocs diminue le nombre de transferts de données, ce qui devient avantageux pour les grandes tailles de matrices.

- Pour les matrices petites à moyennes (`n=512` ou `n=1024`), des tailles de blocs plus petites (`szBlock=32` ou `szBlock=64`) sont optimales.
- Pour les grandes matrices (`n=2048`), des blocs plus grands (`szBlock=1024`) sont préférables.

1.7. Parallélisation du produit par bloc avec OpenMP

Pour paralléliser le produit matrice–matrice par bloc, nous avons utilisé OpenMP en appliquant la directive `#pragma omp parallel for` aux boucles principales du calcul. Nous avons mesuré les temps d'exécution pour différentes valeurs de `OMP_NUM_THREADS`, ainsi que l'accélération parallèle obtenue.

Nombre de threads	Temps (s)	MFlops	Accélération
1	8.99519	239.999	1.000
2	6.67109	321.909	1.348
4	3.77367	569.071	2.384
8	3.12208	687.838	2.881

TABLE 1.5 : Temps d'exécution et accélération du produit matrice–matrice par bloc parallélisé

Analyse des résultats

L'accélération parallèle obtenue avec le produit matrice–matrice par bloc montre une amélioration significative par rapport à la version scalaire parallélisée. Pour un nombre de threads élevé (`OMP_NUM_THREADS = 8`), l'accélération atteint près de 2,88, ce qui indique une bonne utilisation des ressources matérielles.

Cependant, on remarque que l'accélération n'est pas parfaitement linéaire, ce qui peut s'expliquer par :

- **Conflits de cache** : Lorsque plusieurs threads accèdent simultanément à des données dans des blocs proches, des conflits peuvent apparaître, réduisant l'efficacité.
- **Surcharge liée à la synchronisation** : Le coût de synchronisation entre les threads augmente avec leur nombre, ce qui limite les gains de performance.
- **Granularité des blocs** : La taille des blocs (`szBlock = 32`) joue un rôle important dans l'équilibre entre calculs et accès mémoire. Une granularité trop fine ou trop grossière peut entraîner une inefficacité.

Comparaison avec la version scalaire parallélisée

Par rapport à la version scalaire parallélisée, la version par bloc montre une meilleure performance pour tous les nombres de threads. Cela s'explique principalement par une meilleure exploitation de la localité des données dans la version par bloc. En effet, les calculs sont effectués sur de petits sous-blocs qui tiennent mieux dans le cache, réduisant ainsi les temps d'accès à la mémoire principale.

1.8. Comparaison avec BLAS

Problème : Je peux compiler le fichier `test_product_matrice_blas.cpp` avec succès, mais il n'y a aucune sortie dans le résultat.

2. Parallélisation MPI

2.1. Circulation d'un jeton dans un anneau

Objectif : Implémenter un programme qui fait circuler un jeton dans un anneau de processus en utilisant MPI.Le fichier de code de 2.1 se trouve dans token.ring.c.

Algorithme :

1. Le processus de rang zéro initialise un jeton à 1 puis l'envoie au processus de rang 1.
2. Chaque processus de rang p ($0 < p < \text{nbp} - 1$) reçoit le jeton, l'incrémente de 1, puis l'envoie au processus de rang $p + 1$.
3. Le processus de rang nbp $- 1$ reçoit le jeton, l'incrémente de 1, puis l'envoie au processus de rang zéro.
4. Le processus de rang zéro reçoit le jeton du processus de rang nbp $- 1$ et l'affiche à l'écran.

Commande et sortie :

```
gcc -o token_ring token_ring.c -I"D:/ENSTA/OS202_parallel/msmpisdk/Include" \
    -L"D:/ENSTA/OS202_parallel/msmpisdk/Lib/x64" -lmsmpi

mpiexec -n 4 token_ring.exe
Process 0 initializing token with value 1
Process 0 received token with value 4
Process 3 received token and incremented it to 4
Process 1 received token and incremented it to 2
Process 2 received token and incremented it to 3
```

Analyse :

Ce programme démontre le passage d'un jeton à travers un anneau de processus. Chaque processus incrémente la valeur du jeton avant de le transmettre au processus suivant. Ce modèle est utile pour comprendre les bases de la communication point à point en MPI.

2.2. Calcul très approché de π

On veut calculer la valeur de π 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 π comme $\pi = 4 \cdot r$.

L'erreur faite sur π décroît quand le nombre de points générés augmente.

Expériences et résultats :

OpenMP :
MPI :
mpi4py :

Nombre de threads	Temps (s)	Accélération S
1	0.274	1.00
2	0.165	$\frac{0.274}{0.165} \approx 1.66$
4	0.096	$\frac{0.274}{0.096} \approx 2.85$
8	0.086	$\frac{0.274}{0.086} \approx 3.19$

TABLE 2.1 : Temps et accélération avec OpenMP.

Nombre de processus	Temps d'exécution (s)	Accélération S
1	0.296	1.00
2	0.180	$\frac{0.296}{0.180} \approx 1.64$
4	0.106	$\frac{0.296}{0.106} \approx 2.79$
8	0.096	$\frac{0.296}{0.096} \approx 3.08$

TABLE 2.2 : Temps et accélération avec MPI.

Nombre de processus	Temps d'exécution (s)	Accélération S
1	1.446	1.00
2	0.814	$\frac{1.446}{0.814} \approx 1.78$
4	0.600	$\frac{1.446}{0.600} \approx 2.41$
8	0.548	$\frac{1.446}{0.548} \approx 2.64$

TABLE 2.3 : Temps et accélération avec mpi4py.

Analyse :

Les résultats montrent que la parallélisation améliore significativement le temps de calcul pour toutes les approches (OpenMP, MPI et mpi4py). L'accélération obtenue est cohérente avec l'architecture de la machine utilisée et confirme l'indépendance des tâches dans l'algorithme stochastique.

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 hypercube 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.

Commandes et résultats d'exécution :

```
mpiexec -n 2 ./hypercube1d
Task 1 received token 42 from task 0
Task 0 sent token 42 to task 1

mpiexec -n 4 ./hypercube2d
```

```
Task 1 received token 42 from task 0
Task 0 sent token 42 to tasks 1 and 2
Task 2 received token 42 from task 0
Task 3 received token 42 from tasks 1 and 2

mpiexec -n 8 ./hypercube3d
Task 5 received token from task 4
Task 5 sent token to task 7
Task 7 received token from task 5
Task 7 sent token to task 3
Task 3 received token from task 7
Task 3 sent token to task 2
Task 2 received token from task 3
Task 2 sent token to task 6
Task 6 received token from task 2
Task 6 sent token to task 4
Task 4 received token from task 6
```

Temps d'exécution :

- Exécution parallèle 4D : 0.000366 secondes
- Exécution séquentielle 4D : 0.004854 secondes

Calcul de l'accélération :

$$S = \frac{0.004854}{0.000366} \approx 13.26$$

L'accélération obtenue montre une amélioration significative grâce à la parallélisation, exploitant efficacement la topologie de l'hypercube pour minimiser les communications entre nœuds.

A. Annexes

A.1. Code pour la circulation d'un jeton dans un anneau (2.1)

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

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

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        token = 1;
        printf("Process %d initializing token with value %d\n", rank, token);
        MPI_Send(&token, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&token, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received token with value %d\n", rank, token);
    } else {
        MPI_Recv(&token, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        token++;
        printf("Process %d received token and incremented it to %d\n", rank, token);
        if (rank < size - 1) {
            MPI_Send(&token, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        } else {
            MPI_Send(&token, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
    return 0;
}
```

Listing 1 : Circulation d'un jeton dans un anneau (token.ring.c)

A.2. Code pour le calcul de π avec OpenMP (2.2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define NUM_POINTS 10000000

int main() {
    int inside_circle = 0;
    double x, y;
    double pi_estimate;

    double start_time = omp_get_wtime();

    #pragma omp parallel for private(x, y) reduction(+:inside_circle)
    for (int i = 0; i < NUM_POINTS; i++) {
        x = (double)rand() / RAND_MAX * 2.0 - 1.0;
```

```
        y = (double)rand() / RAND_MAX * 2.0 - 1.0;
        if (x*x + y*y <= 1.0) {
            inside_circle++;
        }
    }

    pi_estimate = 4.0 * (double)inside_circle / NUM_POINTS;

    double end_time = omp_get_wtime();

    printf("Estimation de Pi avec OpenMP: %lf\n", pi_estimate);
    printf("Temps d'ex cution: %lf secondes\n", end_time - start_time);

    return 0;
}
```

Listing 2 : Estimation de π avec OpenMP (pi_openmp.c)

A.3. Code pour le calcul de π avec MPI (2.2)

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

#define NUM_POINTS 10000000

int main(int argc, char** argv) {
    int rank, size, local_inside = 0, global_inside;
    double x, y, pi_estimate;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int points_per_proc = NUM_POINTS / size;
    double start_time = MPI_Wtime();

    for (int i = 0; i < points_per_proc; i++) {
        x = (double)rand() / RAND_MAX * 2.0 - 1.0;
        y = (double)rand() / RAND_MAX * 2.0 - 1.0;
        if (x*x + y*y <= 1.0) {
            local_inside++;
        }
    }

    MPI_Reduce(&local_inside, &global_inside, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
        ;

    if (rank == 0) {
        pi_estimate = 4.0 * (double)global_inside / NUM_POINTS;
        double end_time = MPI_Wtime();
        printf("Estimation de Pi avec MPI: %lf\n", pi_estimate);
        printf("Temps d'ex cution: %lf secondes\n", end_time - start_time);
    }

    MPI_Finalize();
    return 0;
}
```

Listing 3 : Estimation de π avec MPI (pi_mpi.c)

A.4. Code pour la diffusion d'un entier dans un réseau hypercube (2.3)

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

int main(int argc, char** argv) {
    int rank, size, token;
    int dim, steps, mask;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    dim = log2(size);
    if ((1 << dim) != size) {
        if (rank == 0) {
            printf("Error: Number of processes must be a power of 2.\n");
        }
        MPI_Finalize();
        exit(0);
    }

    double start_time = MPI_Wtime();
    token = 0;
    if (rank == 0) {
        token = 42;
        printf("Task %d initialized token with value %d\n", rank, token);
    }

    for (steps = 0; steps < dim; steps++) {
        mask = 1 << steps;
        int neighbor = rank ^ mask;

        if (rank < neighbor) {
            MPI_Send(&token, 1, MPI_INT, neighbor, 0, MPI_COMM_WORLD);
            printf("Task %d sent token to task %d\n", rank, neighbor);
        } else {
            MPI_Recv(&token, 1, MPI_INT, neighbor, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
            printf("Task %d received token from task %d\n", rank, neighbor);
        }
    }

    double end_time = MPI_Wtime();
    if (rank == 0) {
        double elapsed_time = end_time - start_time;
        printf("Elapsed time for hypercube diffusion: %f seconds\n", elapsed_time);
    }

    MPI_Finalize();
    return 0;
}
```

Listing 4 : Diffusion d'un entier dans un hypercube (hypercube_d.c)