OS202 – Systèmes Parallèles

TD1 : Produit matrice–matrice et parallélisme

MENESES GAMBOA Carlos

École Nationale des techniques avancées
Janvier 2026

Table des matières

1 Résumé

Ce rapport présente l'analyse des optimisations appliquées au produit matriciel (réorganisation des boucles, parallélisation OpenMP, découpage par blocs) et la mise en œuvre de calculs parallèles avec MPI (approximation de π par Monte Carlo, diffusion hypercube).

2 Machine utilisée

TABLE 1 – Caractéristiques de la machine.

Champ	Valeur
Architecture	x86_64
Model name	11th Gen Intel Core i5-1135G7 @ 2.40GHz
CPU(s)	8
Thread(s) per core	2
Core(s) per socket	4
L1d / L1i cache	192 KiB / 128 KiB
L2 cache	5 MiB
L3 cache	8 MiB
Hypervisor	Microsoft (WSL2)

3 Produit matrice–matrice

3.1 Q1.1 Ordre i-k-j

Le temps d'exécution dépend principalement des accès mémoire, pas de N. Pour $N = 1024$, les *conflict misses* augmentent car le stride est une puissance de 2 exacte, ce qui concentre les accès sur peu de lignes de cache.

TABLE 2 – Série A (Ordre i-k-j).

n	Temps (s)	MFlops
1023	3.68162	581.591
1024	11.4567	187.444
1025	3.99476	539.151

Conclusion : La mémoire est le goulot d'étranglement principal. Les puissances de 2 génèrent des conflits de cache.

3.2 Q1.2 Ordre j-k-i

En réorganisant les boucles pour accéder à des emplacements mémoire contigus (la matrice est stockée en *column-major*), on réduit drastiquement le temps.

Table 3 – Série B (Ordre j-k-i).

n	Temps (s)	MFlops
1023	0.808292	2649.04
1024	0.823736	2607.00
1025	0.836792	2573.86

Conclusion : L'ordre j-k-i est ∼4–5× plus rapide grâce à une meilleure localité spatiale.

3.3 Q1.3 Parallélisation OpenMP

On parallélise la boucle externe (j) pour éviter les *race conditions*.

Table 4 – OMP parallelisation (2, 4 et 8 threads).

Threads	n	Temps (s)	MFlops	Acc.	Effic. (%)
2	1023	0.497505	4303.87	1.62	81.00
2	1024	0.526290	4080.42	1.57	78.50
2	1025	0.511851	4207.83	1.63	81.50
4	1023	0.292308	7325.15	2.77	69.25
4	1024	0.965350	2224.57	0.85	21.25
4	1025	0.275946	7805.08	3.03	75.75
8	1023	0.219521	9753.96	3.68	46.00
8	1024	0.219494	9783.79	3.75	46.88
8	1025	0.296072	7274.52	2.83	35.38

Conclusion : L'efficacité décroît avec le nombre de threads à cause de la contention mémoire et du surcoût OpenMP. Ceci est cohérent avec notre machine (cf. Table 1) : 4 cœurs physiques et 8 threads logiques. Au-delà de 4 threads, l'hyperthreading et la contention cache réduisent l'efficacité.

3.4 Q1.4 Limites

La triple boucle reste limitée par la mémoire. Le *blocking* (découpage par blocs) améliore la réutilisation des données en cache.

3.5 Q1.5-6 Produit par blocs

Table 5 – Produit par blocs (1 thread).

szBlock	n	Temps (s)	MFlops	Acc.
32	1023	0.710867	3012.09	1.14
32	1024	0.702809	3055.57	1.17
32	1025	0.688124	3129.93	1.22
64	1023	0.647724	3305.73	1.25
64	1024	0.662811	3239.96	1.24
64	1025	0.717026	3003.77	1.17
128	1023	0.604135	3544.24	1.34
128	1024	0.646447	3321.98	1.27
128	1025	0.643256	3348.25	1.30

Conclusion : Des blocs plus grands (128) améliorent les performances en réduisant les accès mémoire.

3.6 Q1.7 Blocs parallélisés

TABLE 6 – Parallélisation par blocs.

Thr	szB	n	Temps (s)	MFlops	Acc.	Eff. (%)
2	32	1023	0.400436	5347.17	1.78	89.00
2	32	1024	0.424843	5054.77	1.65	82.50
2	32	1025	0.395033	5452.16	1.74	87.00
2	64	1023	0.393704	5438.60	1.65	82.50
2	64	1024	0.386753	5552.59	1.71	85.50
2	64	1025	0.468303	4599.12	1.53	76.50
2	128	1023	0.412694	5188.35	1.46	73.00
2	128	1024	0.368354	5829.94	1.75	87.50
2	128	1025	0.420138	5126.36	1.53	76.50
4	32	1023	0.332062	6448.19	2.14	53.50
4	32	1024	0.364710	5888.20	1.93	48.25
4	32	1025	0.320053	6729.46	2.15	53.75
4	64	1023	0.310567	6894.47	2.09	52.25
4	64	1024	0.274974	7809.77	2.41	60.25
4	64	1025	0.296754	7257.80	2.42	60.50
4	128	1023	0.253607	8442.98	2.38	59.50
4	128	1024	0.287584	7467.33	2.25	56.25
4	128	1025	0.288379	7468.58	2.23	55.75
8	32	1023	0.251549	8512.06	2.83	35.38
8	32	1024	0.236065	9097.01	2.98	37.25
8	32	1025	0.252851	8518.00	2.72	34.00
8	64	1023	0.220378	9716.04	2.94	36.75
8	64	1024	0.214085	10031.00	3.10	38.75
8	64	1025	0.215673	9986.33	3.32	41.50
8	128	1023	0.200204	10695.10	3.02	37.75
8	128	1024	0.190892	11249.70	3.39	42.38
8	128	1025	0.215779	9981.41	2.98	37.25

Table 7 – Blocs parallélisés vs scalaire parallélisé.

Thr	szB	n	Temps (s)	MFlops	Acc.
2	32	1023	0.406521	5267.13	1.22
2	32	1024	0.411978	5212.61	1.28
2	32	1025	0.454003	4743.98	1.13
2	64	1023	0.447274	4787.22	1.11
2	64	1024	0.375225	5723.19	1.40
2	64	1025	0.433773	4965.23	1.18
2	128	1023	0.346354	6182.10	1.44
2	128	1024	0.348088	6169.37	1.51
2	128	1025	0.393669	5471.04	1.30
4	32	1023	0.277468	7716.93	1.05
4	32	1024	0.287934	7458.24	3.35
4	32	1025	0.294804	7305.81	0.94
4	64	1023	0.284237	7533.15	1.03
4	64	1024	0.268247	8005.62	3.60
4	64	1025	0.282484	7624.45	0.98
4	128	1023	0.254288	8420.36	1.15
4	128	1024	0.251381	8542.74	3.84
4	128	1025	0.275581	7815.42	1.00
8	32	1023	0.237343	9021.53	0.92
8	32	1024	0.232497	9236.60	0.94
8	32	1025	0.267936	8038.41	1.11
8	64	1023	0.246132	8699.38	0.89
8	64	1024	0.226411	9484.87	0.97
8	64	1025	0.228827	9412.27	1.29
8	128	1023	0.220686	9702.49	0.99
8	128	1024	0.206100	10419.60	1.06
8	128	1025	0.206340	10438.00	1.43

Conclusion : Pour petits blocs (32), l'overhead de gestion (boucles supplémentaires, synchronisation OpenMP) domine le gain de localité cache, rendant le blocking moins efficace que le scalaire parallélisé (Acc. < 1). Avec `szBlock` $= 128$, le travail par bloc est suffisant pour amortir l'overhead et exploiter la réutilisation des données en cache L2/L3 (cf. Table 1 : L2=5 MiB, L3=8 MiB), d'où Acc. > 1.

3.7 Q1.8 Comparaison avec BLAS

Table 8 – BLAS vs blocs (1 thread, szBlock=128).

n	BLAS (s / MFlops)	Bloc 1T (s / MFlops)	Acc.
1023	0.886263 / 2415.98	0.604135 / 3544.24	1.47
1024	0.793160 / 2707.50	0.646447 / 3321.98	1.23
1025	0.709822 / 3034.26	0.643256 / 3348.25	1.10

Table 9 – BLAS vs blocs parallélisés (8 threads, szBlock=128).

n	BLAS (s / MFlops)	Bloc 8T (s / MFlops)	Acc.
1023	0.757048 / 2828.35	0.200204 / 10695.10	3.78
1024	0.862041 / 2491.16	0.190892 / 11249.70	4.52
1025	0.751855 / 2864.62	0.215779 / 9981.41	3.48

Conclusion : Notre version par blocs est plus rapide que BLAS même en séquentiel (Acc. > 1). Avec 8 threads, l'accélération atteint $\sim$3.5–4.5$\times$ par rapport à BLAS mono-thread.

4 Parallélisation MPI

4.1 Jeton dans un anneau

Le processus 0 initialise un jeton à 1 et l'envoie au processus 1. Chaque processus p reçoit le jeton, l'incrémente de 1, puis l'envoie au processus $(p+1) \mod \texttt{nbp}$. Le processus 0 reçoit finalement le jeton du processus $\texttt{nbp}-1$ et l'affiche. La valeur finale est `nbp` (nombre de processus), ce qui valide la communication point-à-point en anneau.

4.2 Calcul approché de π

4.2.1 Version OpenMP

TABLE 10 – Monte Carlo π (OpenMP, 40M échantillons).

Threads	Temps (s)	π	Acc.	Effic. (%)
1	0.670217	3.1417421	1.00	100.00
2	0.335759	3.1417264	2.00	100.00
4	0.241612	3.1417286	2.77	69.25
8	0.178217	3.1420238	3.76	47.00
16	0.221487	3.1416330	3.03	18.94

Conclusion : L'accélération plafonne à 8 threads puis décroît. Ceci est cohérent avec notre machine (cf. Table 1) : 4 cœurs physiques $\times$ 2 threads/cœur $=$ 8 threads logiques. À 16 threads, le temps augmente (0.221s vs 0.178s) car les threads supplémentaires génèrent de la sursouscription, des changements de contexte coûteux et de la contention sur les caches partagés.

4.2.2 Version MPI C++

TABLE 11 – Monte Carlo π (MPI C++, 40M échantillons).

Processus	Temps (s)	π	Mop/s
1	0.566340	3.1413573	141
2	0.335046	3.1415713	239
4	0.208942	3.1415776	383
8	0.141896	3.1414698	564
16	0.286767	3.1412587	279

Conclusion : Performances similaires à OpenMP. À 16 processus, le temps double (0.287s vs 0.142s) à cause de la sursouscription : chaque processus MPI rivalise pour les mêmes ressources CPU/mémoire.

4.2.3 Version MPI Python

TABLE 12 – Monte Carlo π (MPI Python, 40M échantillons).

Processus	Temps (s)	π	Mop/s
1	1.277096	3.1410977	62.6
2	0.444834	3.1415155	179.8
4	0.481187	3.1412193	166.3
8	0.474194	3.1416912	168.7
16	0.592600	3.1412151	135.0

Conclusion : Python est ∼3–4× plus lent que C++. Le scaling est mauvais (temps quasi-constant de 4 à 8 proc.) à cause de l'overhead Python. À 16 processus, la sursouscription dégrade encore les performances.

4.3 Diffusion hypercube

TABLE 13 – Accélération hypercube vs séquentiel.

d	Proc.	T_{seq} (s)	T_{hyper} (s)	Acc.
1	2	0.000012	0.000010	1.20
2	4	0.000029	0.000012	2.42
3	8	0.000048	0.000022	2.18
4	16	0.015723	0.000585	26.88

Conclusion : L'hypercube réduit la latence en $O(\log_2 p)$ étapes. Pour $d = 4$ (16 proc.), l'accélération atteint 26.88× : le temps séquentiel explose (15.7ms) alors que l'hypercube reste rapide (0.58ms), démontrant l'efficacité de l'algorithme pour la diffusion à grande échelle.

5 Conclusion générale

— **Mémoire = goulot d'étranglement :** La réorganisation des boucles (j-k-i) améliore la localité spatiale (∼3–4× plus rapide). Le blocking apporte un gain additionnel de ∼1.3×.

— **Parallélisation :** OpenMP et MPI offrent des accélérations jusqu'à ∼3–4× avec 8 threads/processus. L'efficacité décroît au-delà du nombre de cœurs physiques (sursouscription).

— **BLAS :** Notre implémentation par blocs dépasse BLAS (non optimisé sur cette machine). Avec 8 threads, l'accélération atteint ∼3.5–4.5×.

— **Python vs C++ :** L'overhead de l'interpréteur Python multiplie par 3–4× le temps d'exécution par rapport à C++.

— **Hypercube :** Algorithme efficace pour la diffusion avec complexité $O(\log_2 p)$. À 16 processus, l'accélération atteint 27× par rapport à la diffusion séquentielle.