Master Calcul Haut Performance et Simulation

Rapport projet AOC

Thème :

NPB BT OMP

Réalisé par :

M^r.Hery ANDRIANANTENAINA
M^r. Rami BEN-AYED
M^r. Hammid RAMDANI

Encadre par :
M^r. Emmanuel OSERET

Année 2021-2022

Table des matières

1 Présentation du sujet

1.1 Domaine de la miniapp

Le NAS parallel Benchmark(NPB) est un outil qui sert à évaluer la performance des supercalculateurs hautement parallèles. Ces benchmarks consistent en cinq noyaux parallèles et trois benchmarks d'applications simulées. Ensemble, ils imitent les caractéristiques de calcul et de mouvement des données des applications de dynamique des fluides computationnelle (CFD) à grande échelle.

Le principal trait distinctif de ces benchmarks est leur spécification "crayon et papier" : tous les détails de ces benchmarks sont spécifiés uniquement de manière algorithmique. De cette façon, de nombreuses difficultés associées aux approches conventionnelles de benchmarking sur des systèmes hautement parallèles sont évitées.

À cette fin, un certain nombre de noyaux relativement simples ont été conçus. Cependant, les noyaux seuls sont insuffisants pour évaluer complètement le potentiel de performance d'une machine parallèle sur des applications scientifiques réelles. La principale difficulté est qu'une certaine structure de données peut être très efficace sur un certain système pour un des noyaux isolés, et pourtant cette structure de données serait inappropriée si elle était incorporée dans une application plus importante. En d'autres termes, les performances d'une application réelle de dynamique des fluides numérique (CFD) sur un système parallèle dépendent de manière critique du mouvement des données entre les noyaux de calcul.

Le jeu de benchmarks se compose donc de deux éléments principaux : cinq benchmarks de noyaux parallèles et trois benchmarks d'applications simulées. Les benchmarks

d'applications simulées combinent plusieurs calculs d'une manière qui ressemble à l'ordre réel d'exécution dans certains codes d'applications CFD importants.

1.2 Organisation du code

Les huit benchmarks proposés sont :

- ☐ IS - Integer Sort, random memory access
- ☐ EP - Embarrassingly Parallel
- ☐ CG - Conjugate Gradient
- ☐ MG - Multi-Grid
- ☐ FT - discrete 3D fast Fourier Transform
- ☐ BT - Block Tri-diagonal solver
- ☐ SP - Scalar Penta-diagonal solver
- ☐ LU - Lower-Upper Gauss-Seidel solver

Ces différents benchmark possèdent aussi de nombreuse classe dont chaque classe a sa propre taille pour la résolution de chaque problème.

- ☐ Class S : pour des tests rapide et petite
- ☐ Class W : workstation size
- ☐ Classes A, B, C : standard test problems
- ☐ Classes D, E, F : large test problems

Au niveau des parallélisation, ces application sont parallélisables avec MPI et OPENMP. Dans ce projet on a pris l'application BT parallélisé avec OPENMP et on a choisie de lancer l'application avec les tests de problème standard. On a pris la classe A.

La version BT-MZ a été aussi choisi, qui est conçue pour exploiter plusieurs niveaux de parallélisme dans les applications et pour tester l'efficacité des paradigmes et outils de parallélisation multi-niveaux et hybrides (zones de taille inégale à l'intérieur d'une classe à problèmes, augmentation du nombre de zones au fur et à mesure que la classe à problèmes s'agrandit).

1.3 Block Tri-diagonal solver (BT)

BT est un solveur d'équations différentielles partielles (sous la forme de matrices tridiagonales en bloc) intégré dans un pseudo-programme d'application proposé pour l'évaluation des performances des codes CFD sur des processeurs hautement parallèles. Le pseudo-programme d'application est dépourvu des complexités associées aux programmes d'application CFD réels, permettant ainsi une description plus simple des algorithmes. une description plus simple des algorithmes.

Nous calculons ici la solution de multiples systèmes indépendants d'équations tridiagonales en bloc non diagonalement dominantes, avec une taille de bloc de 5x5.

2

Bilan Performance Séquentielle et Parallèle

Ici on a lancé l'application sans aucune modification et on a récupéré les différentes métriques et amélioration que MAQAO nous a fournie lors de l'analyse du programme, en comparant la version séquentielle avec celle parallèle.

2.1 Caractéristique de la machine

Pour nos tests, on a utilisé la machine knl01 dont les caractéristiques sont résumé par la figure suivante :

Experiment Summary

Application	./bin/bt-mz.A.x		
Timestamp	2022-03-06 11:28:58	**Universal Timestamp**	1646566138
Number of processes observed	1	**Number of threads observed**	1
Experiment Type	Sequential		
Machine	knl01		
Model Name	Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz		
Architecture	x86_64	**Micro Architecture**	KNIGHTS_LANDING
Cache Size	1024 KB	**Number of Cores**	64
OS Version	Linux 5.4.0-100-generic #113-Ubuntu SMP Thu Feb 3 18:43:29 UTC 2022		
Architecture used during static analysis	x86_64	**Micro Architecture used during static analysis**	KNIGHTS_LANDING
Compilation Options	**bt-mz.A.x**: GNU 9.3.0 -mtune=generic -march=x86-64 -g -O3 -fopenmp -fintrinsic-modules-path /usr/lib/gcc/x86_64-linux-gnu/9/finclude -fpre-include=/usr/include/finclude /math-vector-fortran.h		

FIGURE 2.1 – Caractéristiques de la machine

Cette machine est équipée par un processeur Intel Knights Landing avec 64 coeurs et 256 threads.

2.2 Mesure de performance

Avant de commencer les mesures de performances, il faut tout d'abord choisir une classe de problème pour benchmark BT entre S, W, A, B, C , D, E, F.

Le problème est que les classes S et W est qu'elles sont limitées par le nombre des threads qu'on peut utiliser et s'exécutent en quelques secondes.

Le problème avec les classes B,C et surtout D, E, F est qu'elles prennent beaucoup du temps à s'exécuter malgré leur support de plusieurs threads (plus de 64).

Donc on a choisi la classe A qui offre un temps d'exécution adéquat mais qui ne supporte que 16 threads.

Global Metrics

Total Time (s)		163.34
Profiled Time (s)		163.34
Time in analyzed loops (%)		47.5
Time in analyzed innermost loops (%)		46.1
Time in user code (%)		99.8
Compilation Options		**bt-mz.A.x**: -march=(target) is missing. -funroll-loops is missing.
Perfect Flow Complexity		1.01
Array Access Efficiency (%)		87.0
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	10
FP Vectorised	Potential Speedup	1.21
	Nb Loops to get 80%	11
Fully Vectorised	Potential Speedup	1.71
	Nb Loops to get 80%	13
FP Arithmetic Only	Potential Speedup	1.40
	Nb Loops to get 80%	11

FIGURE 2.2 – Globals metrics

Les métriques globales de MAQAO nous donnent un temps d'exécution de 163s pour le code originale de BT en version séquentielle. On remarque aussi que le code de base est plus au moins optimisé en ignorant les options de compilation et on regardant les accélération potentielles suggérées par MAQAO.

2.3 Version Parallèle vs Séquetielle

Afin de mieux comprendre la scalabilité du benchmark BT, on a décidé de comparer le temps d'éxecution du programme pour un nombre n des threads entre 1 et 16 (**OMP_NUM_THREADS=n**). La classe A étant limité à 16 threads.

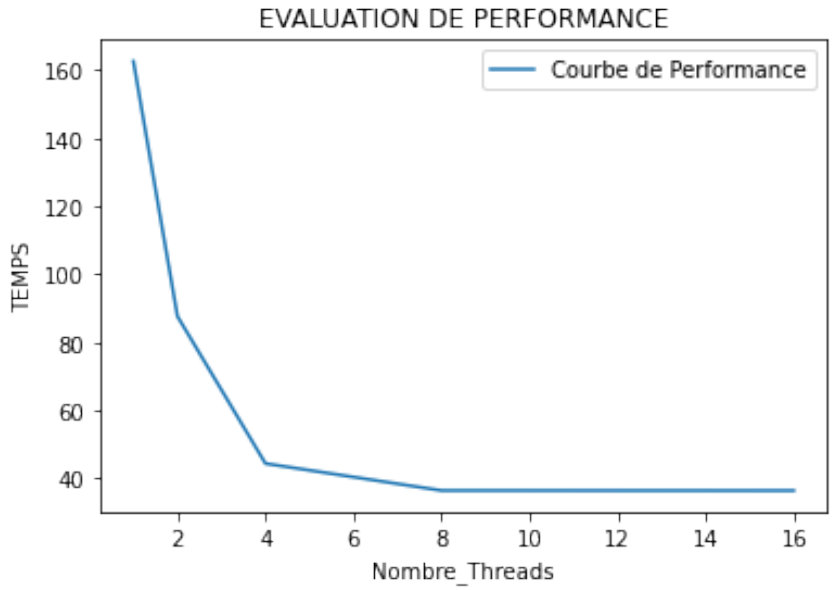


FIGURE 2.3 – Courbe de performance

On remarque que le programme atteint son accélération maximale de 4.87 avec 8 threads, malgré que la classe A supporte 16 threads. Cela suggère qu'il y a une partie du code qui se comporte comme un bottleneck, et qui n'est pas parallèlisable.

Name	Module	Coverage run_0 (%)
○ binvcrhs	bt-mz.A.x	31.39
○ matmul_sub	bt-mz.A.x	15.6
► z_solve_._omp_fn.0	bt-mz.A.x	12.48
► y_solve_._omp_fn.0	bt-mz.A.x	12.16
► x_solve_._omp_fn.0	bt-mz.A.x	11.47
► compute_rhs_._omp_fn.0	bt-mz.A.x	10.74
○ matvec_sub	bt-mz.A.x	3.72
○ binvrhs	bt-mz.A.x	0.73
○ lhsinit	bt-mz.A.x	0.66
► add_._omp_fn.0	bt-mz.A.x	0.34
○ exact_solution	bt-mz.A.x	0.19
► copy_x_face	bt-mz.A.x	0.16

FIGURE 2.4 – Temps d'éxecution des routines

En regardant de prés le rapport MAQAO, on constate que les routines **binvcrhs** et **matmul_sub** sont en fait non parallélisables et prennent une grande partie du temps d'éxecution.

3
Optimisation du Code

3.1 Courbes de vitesse

Avec MAQAO, on peut obtenir les courbes de vitesse qu'on doit avoir après avoir fait tous les modification nécessaire dans l'application. Dans notre cas on a les courbes résumes dans la figure suivante :

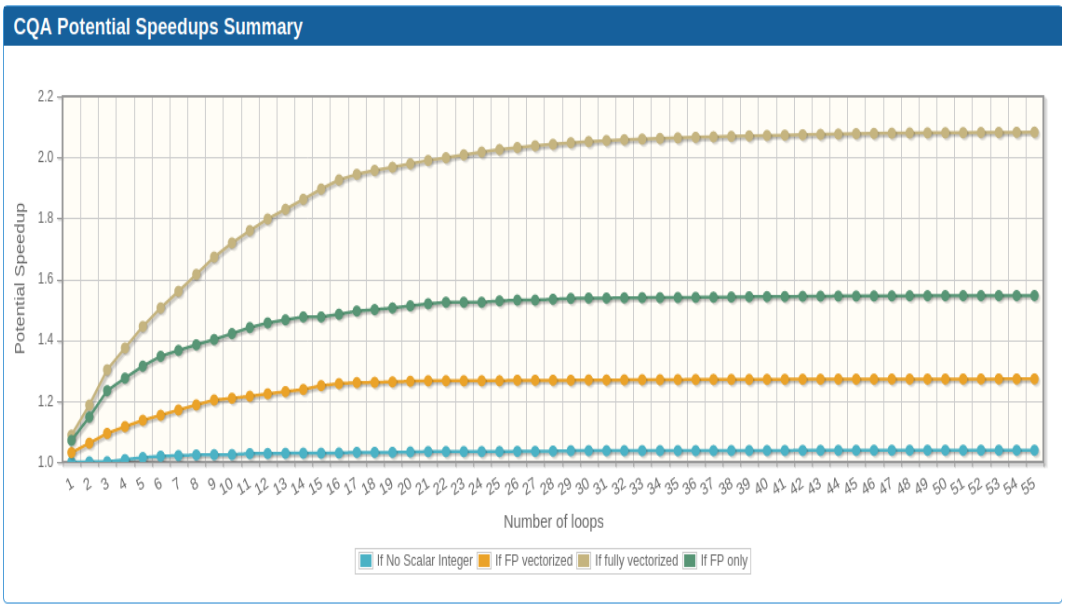


FIGURE 3.1 – Potential speedups

On peut donc déduire, d'après les suggestions de MAQAO, qu'on peut obtenir une accélération potentielle entre 1.2 et 2.1 si l'on vectorise notre code.

3.2 Compilation avec -O3 avec l'option -march=native -funroll-loops

Dans cette partie on a rajouté les options qui manquent dans la compilation et on a gardé l'option d'optimisation de base qui est **-O3**.

Global Metrics

Total Time (s)		147.56
Profiled Time (s)		147.56
Time in analyzed loops (%)		42.2
Time in analyzed innermost loops (%)		39.8
Time in user code (%)		99.9
Compilation Options		OK
Perfect Flow Complexity		1.01
Array Access Efficiency (%)		57.7
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.05
	Nb Loops to get 80%	11
FP Vectorised	Potential Speedup	1.09
	Nb Loops to get 80%	10
Fully Vectorised	Potential Speedup	1.59
	Nb Loops to get 80%	13
FP Arithmetic Only	Potential Speedup	1.34
	Nb Loops to get 80%	12

FIGURE 3.2 – Compilation avec **-O3**

Ici on peut constater qu'au niveau de compilation on est bon. On peut voir aussi l'amélioration du temps d'exécution du programme.

Ci-dessous la courbe de performance issue de la compilation avec l'option de compilation -O3 avec des différents nombre de threads.

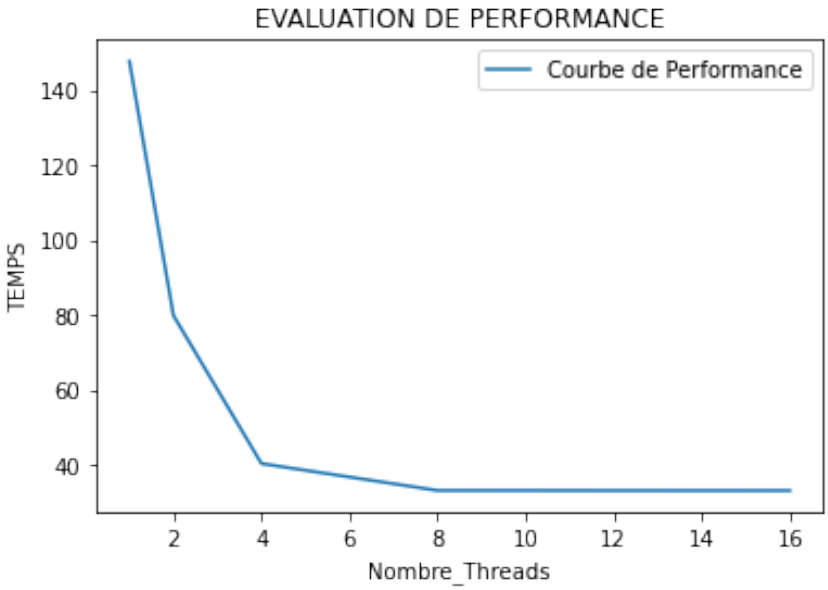


FIGURE 3.3 – Courbe de performance avec **-O3**

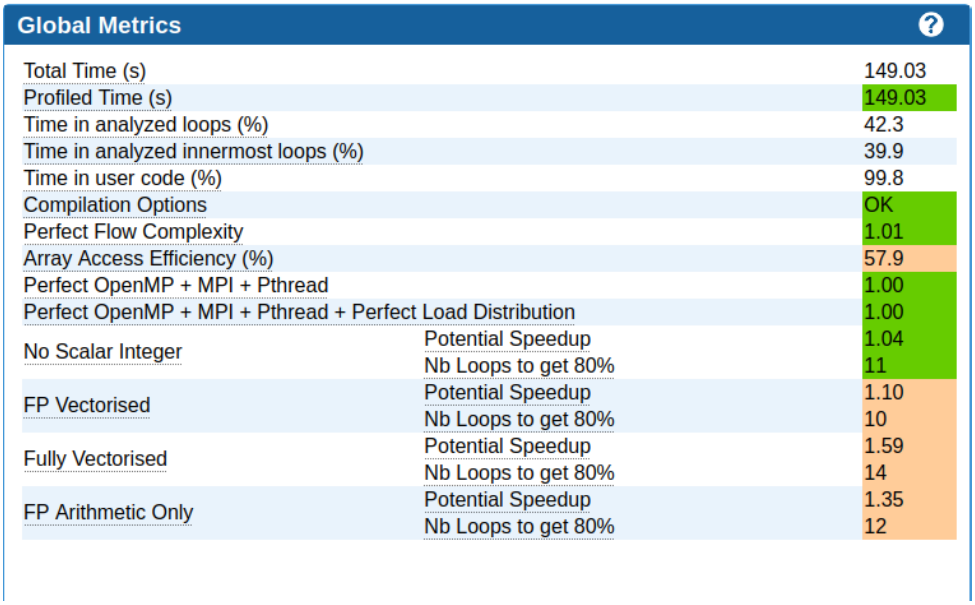
On constate que, malgré les améliorations de performances, le code n'est pas encore convenablement vectorisé en analysant le rapport MAQAO, et les bottlenecks persistent encore.

3.3 Compilation avec -Ofast

Afin d'essayer de forcer la vectorisation du code, on a utilisé de compilation **-Ofast**.

Par rapport à la version base line, on voit une amélioration au niveau du temps d'exécution. Mais on peut constaté qu'en compilant avec **-O3** on a une meilleur performance.

Dans la suite de notre projet on va garde l'option **-O3**.



Global Metrics

Total Time (s)		149.03
Profiled Time (s)		149.03
Time in analyzed loops (%)		42.3
Time in analyzed innermost loops (%)		39.9
Time in user code (%)		99.8
Compilation Options		OK
Perfect Flow Complexity		1.01
Array Access Efficiency (%)		57.9
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.04
	Nb Loops to get 80%	11
FP Vectorised	Potential Speedup	1.10
	Nb Loops to get 80%	10
Fully Vectorised	Potential Speedup	1.59
	Nb Loops to get 80%	14
FP Arithmetic Only	Potential Speedup	1.35
	Nb Loops to get 80%	12

FIGURE 3.4 – Compilation avec **-Ofast**

3.4 Utilisation d'un autre compilateur

Dans cette partie nous allons consacrer notre temps sur les modifications de la structure du code de l'application. Ces changements seront effectuer en suivant les recommandation issue de l'application MAQAO.

Parmi les suggestions que MAQAO nous a fournie, on a utilisé un autre compilateur.

Par rapport à **gcc**, on peut constater su la figure ci-dessus que **icc** offre beaucoup de performance.

Global Metrics		
Total Time (s)		120.04
Profiled Time (s)		120.04
Time in analyzed loops (%)		53.7
Time in analyzed innermost loops (%)		50.7
Time in user code (%)		99.8
Compilation Options		OK
Perfect Flow Complexity		1.01
Array Access Efficiency (%)		57.9
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.06
	Nb Loops to get 80%	11
FP Vectorised	Potential Speedup	1.12
	Nb Loops to get 80%	10
Fully Vectorised	Potential Speedup	1.89
	Nb Loops to get 80%	14
FP Arithmetic Only	Potential Speedup	1.48
	Nb Loops to get 80%	12

FIGURE 3.5 – Compilation avec **-icc**

On constate alors qu'on obtient une amélioration majeure de performance par rapport au code initial, ce qui revient à une accélération de 1.4 (entre 1.2 et 2.1 proposé par MAQAO).

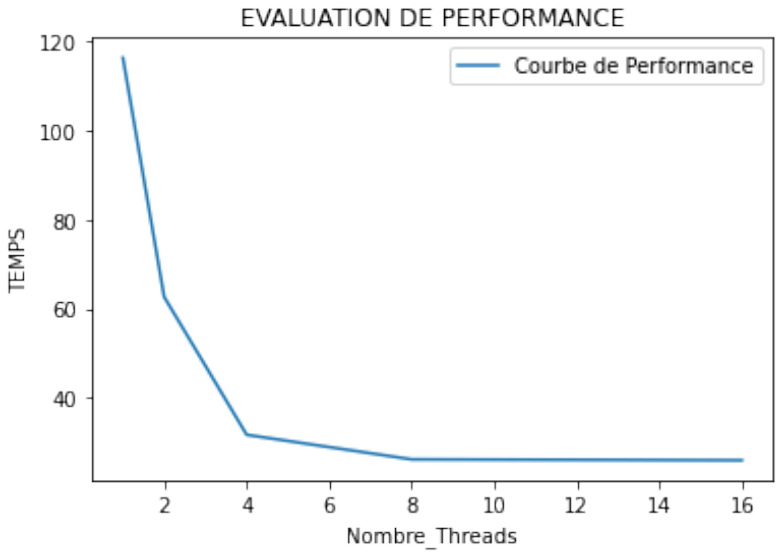


FIGURE 3.6 – Courbe de performance **-icc**

En ce qui concerne les bottlenecks, on observe malheureusement que le code n'est encore pas scalable au delà de 8 threads ce qui est peut être du à la nature du code lui même.

Par contre, on obtient une code bien vectorisé, ce qui est apparent si on analyse la version assembleur du code.

On compare par exemple la routine **binvcrhs** de base et après modifications :

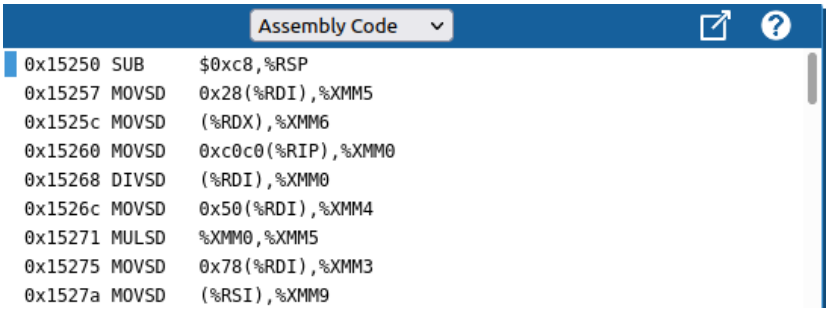


FIGURE 3.7 – la routine binvcrhs avant vectorisation

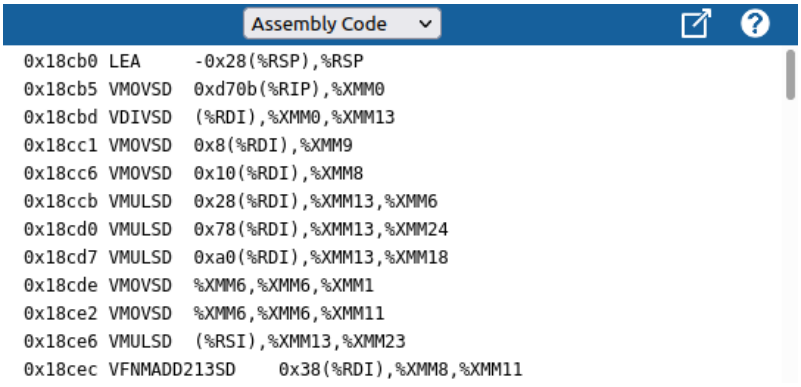


FIGURE 3.8 – la routine binvcrhs après vectorisation