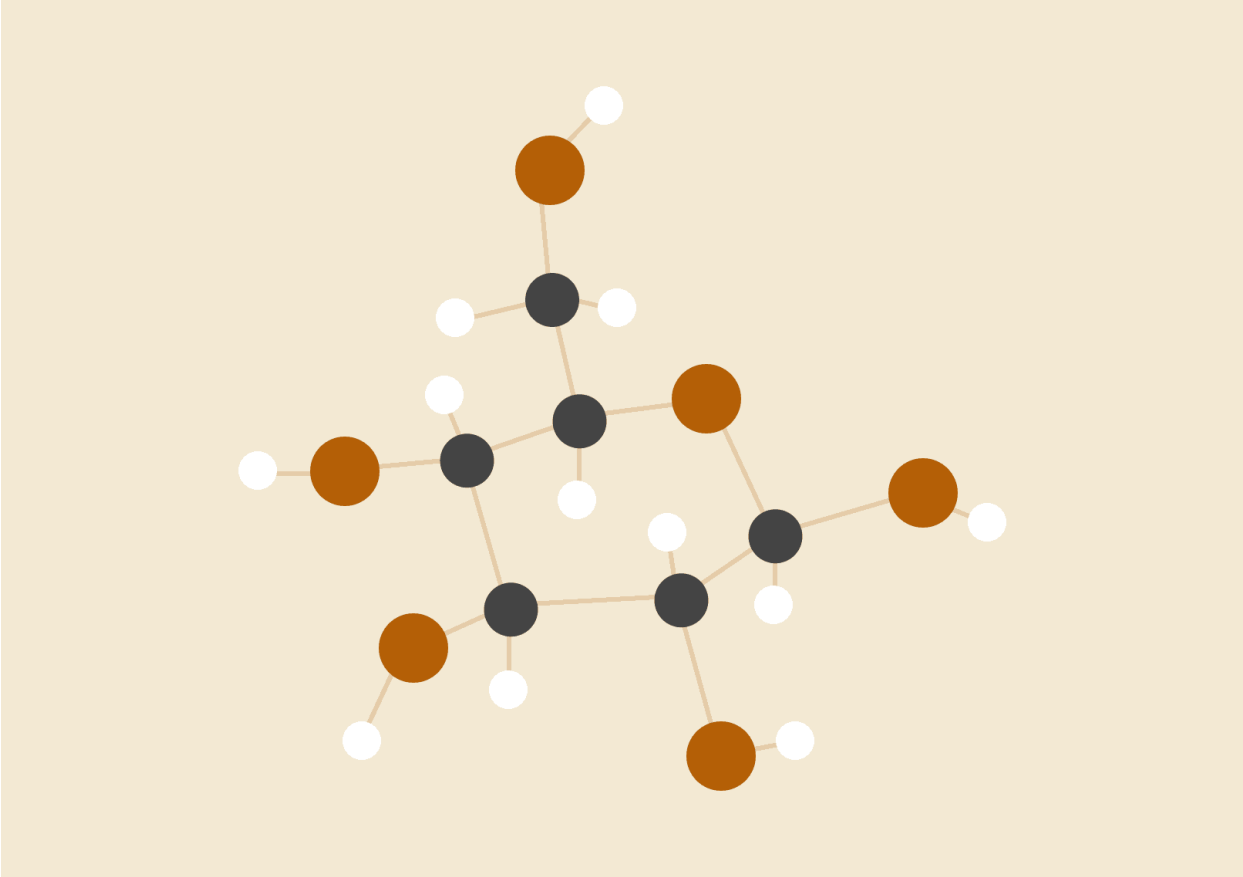
RAPPORT PROJET PPAR FFT

Département d’informatique - Sorbonne Université FSI M1 CCA - Décembre 2023



XIE JAROD (n° étudiant : 28710097)

07/01/2024

INTRODUCTION

Dans ce projet, vous disposez d'un programme séquentiel en langage C, fft.c, qui effectue diverses opérations liées à la Transformée de Fourier Rapide (FFT). Le programme génère du bruit blanc, effectue une transformée de Fourier, applique une transformation aux coefficients de Fourier, effectue une transformée de Fourier inverse, et produit en sortie une partie du résultat sous la forme d'un fichier audio. Ces opérations aboutissent à un son "étouffé", évoquant une expérience sonore sous-marine.

L'objectif principal est d'améliorer les performances de ce programme en développant une version parallèle qui utilise efficacement les ressources de calcul d'un seul nœud de calcul. Pour ce faire nous nous sommes concentrés sur l'utilisation d'OpenMP pour la parallélisation en raison de sa simplicité et de son efficacité dans le calcul parallèle à mémoire partagée.

Les optimisations potentielles incluent la parallélisation de sections de code à l'aide de directives OpenMP, l'utilisation de la vectorisation pour améliorer l'efficacité computationnelle, la gestion soigneuse de la hiérarchie mémoire pour minimiser les déplacements de données, et l'utilisation de bibliothèques externes couramment disponibles dans les centres de calcul.

Tables des matières

1. **Parallélisation avec OpenMP :** Explorez les opportunités de paralléliser des sections du code en utilisant les directives OpenMP. Identifiez et parallélisez des boucles et des tâches intensives en calcul pour exploiter le potentiel de plusieurs cœurs sur un seul nœud de calcul.

2. **Vectorisation :** Envisager des possibilités de vectorisation pour exploiter les instructions SIMD (Single Instruction, Multiple Data). Optimisez les boucles critiques en utilisant des techniques de vectorisation pour améliorer les performances.

3. **Analyse d'efficacité (Benchmark):** Le benchmark quantifie les améliorations de performance entre les versions séquentielle et parallèle du programme. Mesurant le temps d'exécution, ils guident les choix d'optimisation, contribuant à maximiser l'efficacité sur un nœud de calcul unique.

4. **Difficultés rencontrées** : La gestion des difficultés est essentielle dans ce projet. Identifier les zones propices à la parallélisation, assurer une synchronisation appropriée avec OpenMP et optimiser les opérations vectorielles sont des défis critiques. Surmonter ces obstacles nécessitera une expertise approfondie pour maximiser l'efficacité du programme parallèle.

5. **Conclusion :** Ouverture sur d'autres problèmes et possibilités de résolution.

Parallélisation avec OpenMp

L'introduction à la parallélisation avec OpenMP offre une plongée stratégique dans l'amélioration des performances du code en exploitant la puissance des architectures à mémoire partagée. En explorant les opportunités de parallélisation, l'objectif est d'optimiser les sections critiques du code en utilisant les directives OpenMP. En ciblant spécifiquement les boucles et les tâches intensives en calcul, cette approche vise à tirer pleinement parti des capacités de traitement parallèle, permettant une exécution plus rapide sur un seul nœud de calcul. L'efficacité résultante contribuera significativement à l'amélioration globale des performances du programme.

Pour ce faire j'ai rajouté des directives OpenMP (#pragma omp parallel for) pour presque toutes les boucles "for" pertinents du code fournis donc notamment dans les fonctions qu'on appelle abondamment tel que "FFT_rec","FFT","IFFT" et aussi dans le main pour pouvoir générer le bruit blanc, ajuster les coéfficients de Fourier et normaliser le résultat avec une meilleur efficacité.

J'ai de plus ajouté des "#pragma omp task" pour créer des tâches parallèles pour les sous-problèmes de la FFT, ce qui permet une exécution simultanée sur plusieurs cœurs.

La directive "#pragma omp taskwait" est ensuite utilisée pour attendre la fin de l'exécution de toutes les tâches parallèles créées avant de poursuivre. En adoptant cette approche, l'algorithme exploite efficacement les architectures parallèles en distribuant le travail sur plusieurs threads, améliorant ainsi les performances de la FFT pour des tailles plus grandes. Cela contribue à une utilisation optimale des ressources parallèles disponibles, en particulier sur des architectures multicœurs. (fichier : fft_omp_first.c lignes 84 à 89)

```
84  #pragma omp task
85          FFT_rec(n / 2, X, Y, 2 * stride);
86  #pragma omp task
87          FFT_rec(n / 2, X + stride, Y + n / 2, 2 * stride);
88  #pragma omp taskwait
89      }
```

J’ai rajouté dans la fonction FFT la directive #pragma omp single qui garantie que le bloc de code qui suit ne sera exécuté que par un seul thread, généralement le thread principal. Cela évite les problèmes de concurrence lors de l'appel à la fonction récursive FFT_rec, assurant ainsi une exécution correcte de l'algorithme de la FFT.

```
108    #pragma omp parallel
109        {
110    #pragma omp single
111            FFT_rec(n, X, Y, 1); /* stride == 1 initially */
112        }
```

Voyons maintenant un exemple pour le temps d'exécution sur une taille de 2^25. (Effectué sur ma machine)

```
jarod@DESKTOP-B2P9JGS:/mnt/c/Users/Jarod/Desktop/Cours/M1/NEW_PPAR$ ./fft_omp_first --size $((2**25))
Generating white noise...
Forward FFT...
Adjusting Fourier coefficients...
Inverse FFT...
Normalizing output...
max = 3216.44
Temps = 55.038936s
```

Le temps à été calculé en faisant la différence du temps lors de l'appellation de la fonction main jusqu'à la fin de son exécution.

Le temps total est d' environ 55 secondes ce qui est relativement rapide pour une taille aussi grande, le résultat est convainquant. Pour voir la comparaison avec l’algorithme naïf, voir la partie Benchmark.

Vectorisation

La vectorisation, exploitant les instructions SIMD (Single Instruction, Multiple Data), accélère les calculs en traitant simultanément plusieurs données. Ciblant les boucles et opérations intensives, cette approche maximise l'utilisation des unités de traitement vectoriel, réduisant le temps d'exécution sur des architectures compatibles avec la vectorisation.

Pour cela, nous allons introduire une nouvelle structure de données pour pouvoir vectoriser notre problème. La structure de données sépare explicitement les parties réelles et imaginaires des nombres complexes en deux tableaux distincts (real et imag). Cette séparation facilite l'alignement mémoire et l'accès simultané aux parties réelles et imaginaires lors de l'utilisation d'instructions SIMD.

```
30  typedef struct arrayComplex_
31  {
32      double *real;
33      double *imag;
34  } arrayComplex;
```

J'ai dû donc modifier toutes les instances faisant références à complexe pour avoir des données cohérentes avec la nouvelle structure. Par exemple dans FFT_rec et FFT_rec_vect j'ai désormais ces lignes qui font exactement la même chose :

```
double complex p = Y[i];
```

Avant

```
double pReal = YR[i];
double pImg = YI[i];
```

Après

On vient maintenant à la partie la plus technique avec l'introduction des instructions SIMD dans notre code initial. Je vais détailler une partie du code dans cette section, je choisis arbitrairement la partie que j'ai modifiée sur la IFFT_vect. Nous avons utilisé la SIMD pour faire le calcul du conjugué.

J'ai donc créé un vecteur SIMD de double précision, initialisé avec la valeur -1.0. Il est utilisé pour multiplier les parties imaginaires des nombres complexes par -1.0

```
void iFFT_vect(u64 n, arrayComplex *X, arrayComplex *Y)
{
    __m256d vect_minus_one = _mm256_set1_pd(-1.0);
```

Puis dans une boucle for, je multiplie les parties imaginaires par -1.0 à l'aide du vecteur introduit, je récupère dans vect_imag la partie imaginaire et multiplie par vect_minus_one. Puis je restock la valeur du vecteur résultat dans la partie imaginaire concerné.

```
#pragma omp parallel for
    for (u64 i = 0; i < n; i += 4)
    {
        __m256d vect_imag = _mm256_load_pd(&X->imag[i]);
        __m256d vect_res = _mm256_mul_pd(vect_minus_one, vect_imag);
        _mm256_store_pd(&X->imag[i], vect_res);
    }
```

J'appelle ensuite FFT(n, X , Y) avec les parties imaginaires de X qui ont été modifiées. Je fait une nouvelle boucle for pour pouvoir calculer le conjugué des valeurs de Y tout en normalisant par n avec la variable div_n. Comme précédemment, je vais load les données que je veux et les modifier puis les stocker au même emplacement. Ce qui conclut ma fonction IFFT.

```
    __m256d div_n = _mm256_set1_pd(n);

#pragma omp parallel for
    for (u64 i = 0; i < n; i += 4)
    {
        __m256d real_vals = _mm256_load_pd(&Y->real[i]);
        __m256d imag_vals = _mm256_load_pd(&Y->imag[i]);

        real_vals = _mm256_div_pd(real_vals, div_n);
        imag_vals = _mm256_div_pd(imag_vals, div_n);
        imag_vals = _mm256_mul_pd(imag_vals, vect_minus_one);

        _mm256_store_pd(&Y->real[i], real_vals);
        _mm256_store_pd(&Y->imag[i], imag_vals);
    }
}
```

Voyons maintenant un exemple pour le temps d'exécution sur une taille de 2^25. (Effectué sur ma machine)

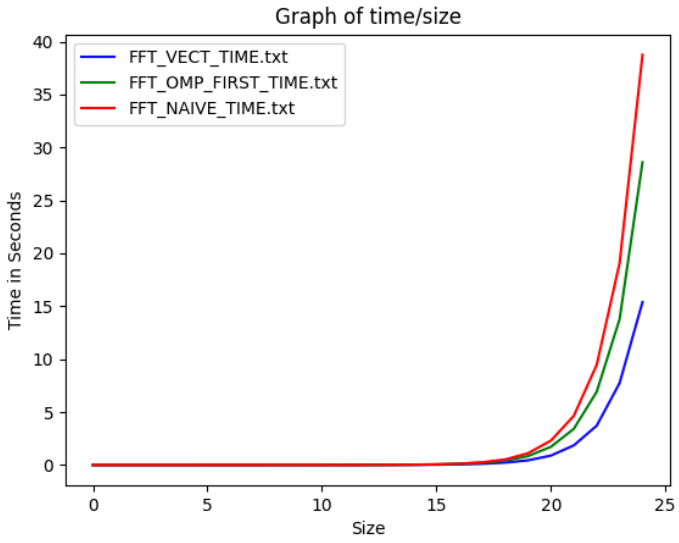
```
jarod@DESKTOP-B2P9JGS:/mnt/c/Users/Jarod/Desktop/Cours/M1/NEW_PPAR$ ./fft_vect --size $((2**25))
Nb thread = 8
Generating white noise...
Forward FFT...
Adjusting Fourier coefficients...
Inverse FFT...
Normalizing output...
max = 3241.79
Temps = 10.097765s
```

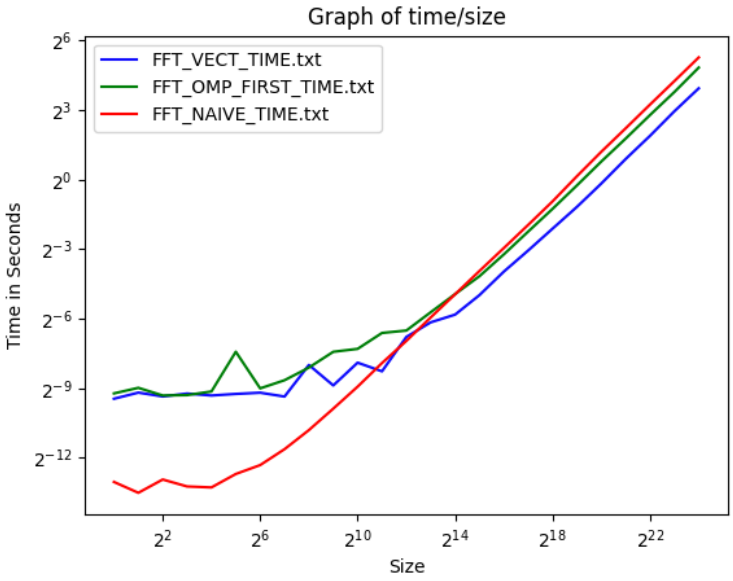
L'exécution à été effectuée en environ 10 secondes, par rapport à la fonction sans vectorisation nous avons un gain de temps potentiel fois 5. Nous pouvons affirmer que notre vectorisation est un succès et qu'il est nettement plus rapide.

BenchMark

L'étape cruciale du benchmarking dans notre projet vise à évaluer l'efficacité des optimisations appliquées au code FFT. En mesurant et comparant les temps d'exécution de différentes implémentations, telles que la parallélisation avec OpenMP et la vectorisation SIMD, nous déterminerons les gains de performances. Ces évaluations guideront nos choix d'optimisations pour maximiser l'efficacité du programme, créant ainsi une version optimale du code FFT.

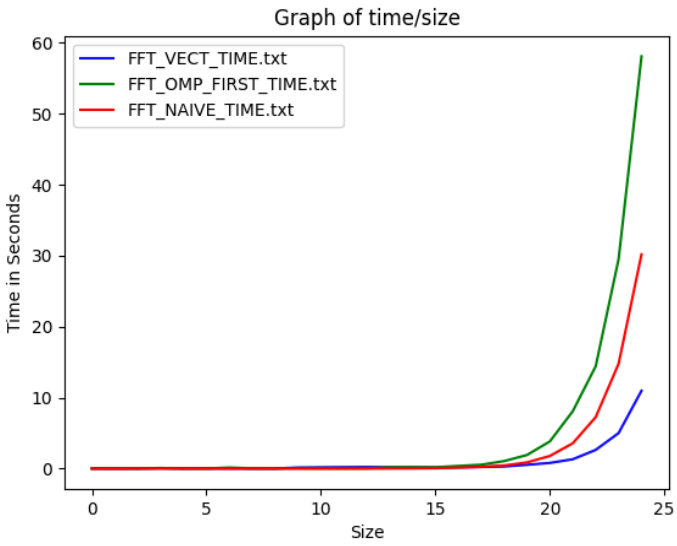
Tous les calculs ont été effectués sur le cluster nommé grisous à Nancy avec 32 threads avec une connexion ssh à distance. Les 2 graphiques prennent les mêmes données en entrée. Le seul changement est la base prise en compte (en base 2 à droite).

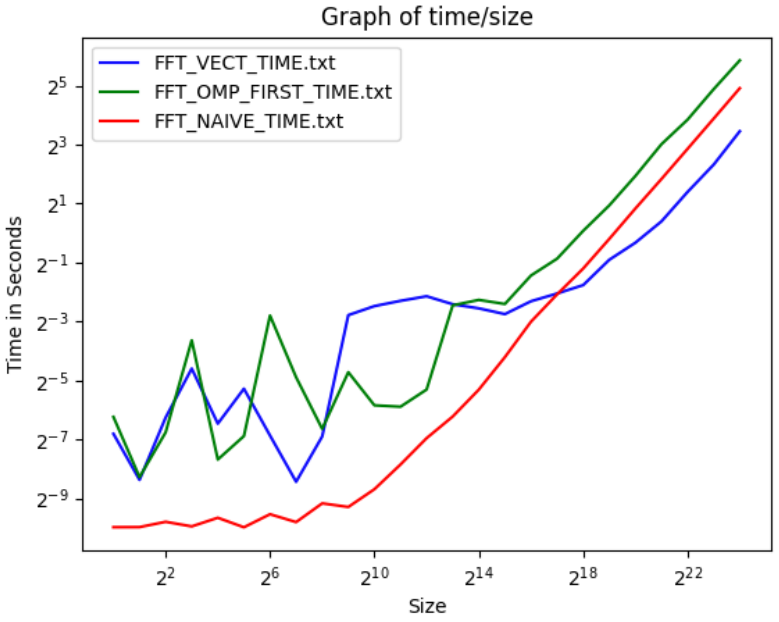




On voit très clairement que nos 2 méthodes sont plus efficaces sur des tailles plus élevées. Et que la version naïve cependant est très rapide sur des tailles moins importantes.

Je vais aussi essayer de tester sur ma propre machine avec 8 threads, pour voir la différence.





On remarque tout d'abord que la version naïve est très rapide sur des tailles moins importantes. De plus, quelque chose d'assez surprenant, la vitesse d'exécution de la méthode avec instructions OpenMP est plus lente que celle naïve pour n'importe quelle taille, pourtant celle vectorielle est toujours plus rapide à partir d'une certaine taille. On remarque aussi que la vitesse d'exécution à été plus rapide sur notre machine avec 8 threads pour la méthode vectorielle que sur Grid5000 avec 32 threads.

On peut conclure que le nombre de threads est un facteur majeur quant à la vitesse de notre méthode "OpenMP only" et qu'elle a un effet négatif pour notre méthodes vectorielle. Cette perte d'efficacité peut être causée par plusieurs facteurs, dont la gestion des ressources matérielles, la concurrence pour l'accès à la mémoire partagée, et la surcharge induite par la parallélisation elle-même. Lorsque le nombre de threads augmente, il peut y avoir une augmentation significative des conflits de mémoire, des temps d'attente pour l'accès aux ressources partagées et une utilisation sous optimale des caches, contribuant ainsi à une dégradation des performances. Il est crucial d'optimiser la gestion des ressources et de minimiser les points chauds du programme pour atténuer ces effets indésirables lors de l'augmentation du nombre de threads.

Difficulté rencontrées

Dans la réalisation de ce projet, certaines difficultés ont émergé, nécessitant une analyse approfondie et des stratégies d'optimisation spécifiques. Ces défis comprennent la gestion efficace de la concurrence dans l'accès aux ressources partagées, la minimisation des conflits de mémoire, et l'optimisation de la vectorisation pour exploiter pleinement les capacités du matériel. Trouver le bon équilibre entre la parallélisation, la vectorisation et la gestion des ressources s'est avéré être un défi complexe.

Les premières difficultés ont résidé dans l'acquisition des directives OpenMP et leur utilisation efficace. Pour garantir leur cohérence et leur utilité, plusieurs essais sur des exemples plus restreints ont été nécessaires afin de maîtriser pleinement ces directives.

Par la suite, le défi majeur s'est présenté lors de la phase de vectorisation. Créer une nouvelle structure de données au départ a constitué une étape non négligeable, suivie de la nécessité de se familiariser avec les instructions SIMD. Introduire ces instructions dans des parties spécifiques du code tout en prenant en compte les directives OpenMP déjà présentes a représenté un défi supplémentaire.

En dépit de ces défis, chaque étape a été abordée avec persévérance et résolution. L'apprentissage progressif des directives OpenMP a permis d'optimiser la parallélisation du code, offrant ainsi des performances accrues sur des tâches intensives en calcul. De même, surmontant les obstacles liés à la vectorisation, la création d'une nouvelle structure de données a permis une meilleure gestion des opérations SIMD, contribuant ainsi à l'amélioration significative des performances du code.

Conclusion

La conclusion de ce projet offre une perspective enrichissante sur les défis et les solutions rencontrés lors de l'optimisation du code FFT. L'exploration des directives OpenMP et la maîtrise de la vectorisation ont constitué des étapes cruciales, témoignant de la complexité inhérente à l'optimisation.

Nous avons examiné les divers aspects positifs et négatifs introduits par la parallélisation avec OpenMP et la vectorisation des données. Les avantages comprennent une amélioration des performances, surtout avec un nombre important de threads pour des données d'entrée substantielles. Cependant, les inconvénients se manifestent par une diminution des performances avec la méthode de vectorisation lorsque le nombre de threads augmente. Il est donc nécessaire de trouver un équilibre optimal entre le nombre de threads et l'utilisation des instructions SIMD

En termes d'ouverture sur d'autres problèmes, cette expérience suggère une réflexion approfondie sur l'impact de la parallélisation et de la vectorisation dans des contextes variés. Les enseignements tirés de ce projet pourraient être appliqués à d'autres algorithmes numériques complexes, offrant ainsi des perspectives intéressantes pour des applications scientifiques et techniques diverses