UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN GÉNIE ÉLECTRIQUE

PAR
MOUNIR KHELIFI

IMPLÉMENTATION PARALLÈLE DES FFTs SUR DES SYSTÈMES MULTICŒURS POUR LA COUCHE PHYSIQUE DU LTE 4G

JUILLET 2016

Résumé

La communication LTE (*Long Term Evolution*) vise à répondre adéquatement aux demandes croissantes de la visioconférence, le vidéo streaming, les applications de jeux, et le transfert rapide dans les applications mobiles. Cependant, son protocole complexe impose des contraintes strictes sur les couches des communications inférieures (couche application et physique) où la mise en œuvre des opérations à forte demande en calcul est un défi. Ceux-ci comprennent la transformation rapide de Fourier (FFT – *Fast Fourier Transform*) et le décodage de canal. Ces opérations de couche physique doivent être accomplies à plusieurs reprises dans un temps de calcul bien précis (66,7 µs) afin de respecter la norme LTE (la latence). Ce temps d'exécution est actuellement assuré par des ressources de calcul comme des processeurs dédiés et FPGA. Ce projet est une contribution aux travaux sur le transfert des calculs en bande de base vers les centres de traitement des données (*Cloud* ou *Centralized Radio Access Network*). Ainsi, le problème de temps d'exécution (latence) est un défi à surmonter afin d'adopter des machines virtuelles et des technologies de pointes respectant le protocole LTE.

L'objectif principal de ce travail est de proposer des implémentations parallèles de FFTs indépendantes respectant les contraintes du protocole LTE basés sur l'OFDM (*Orthogonal Frequency Division Multiplexing*). Plus précisément, il s'agit d'exécuter 100 FFTs indépendantes dans un temps inférieure à 66,7 µs sur des plateformes Multicœur et Manycœur. Nous avons utilisé plusieurs bibliothèques pour ce but comme FFTW (*Fastest*

Fourier Transform of the West), MKL (*Math Kernel Library*), OpenMP (*Open Multiprocessing*), MPI (*Message Passing Interface*) et DPDK (*Data Plane Development Kit*).

Au début, nous avons utilisé DPDK pour exécuter une seule FFT et huit FFTs indépendantes. Cette étape nous a permis d'évaluer le temps de réponse d'une FFT sur des processeurs à quatre et huit threads. Ultérieurement, nous avons proposé une implémentation native et optimisée sur le Xeon-Phi (MIC – *Many Integrated Core*) en utilisant OpenMP. Des performances sont présentées sur les plateformes Intel Core-i7, Xeon et Xeon-Phi. Les meilleurs résultats ont été obtenus avec le Xeon-Phi, mais avec des temps excédant le 66,7 µs, soit un temps de 120 µs pour exécuter 100 FFTs indépendantes.

Dans la troisième section du mémoire, sept implémentations des FFTs ont été comparées ; OpenMP sur 1 MIC du Xeon-Phi , MPI sur 1, 2 et 3 MICs, OpenMP + MPI sur 1 MIC et 3 MICs et MPI sur une plateforme hétérogène de Xeon + 3 MICs. Les résultats expérimentaux ont montré que seulement la combinaison hétérogène de Xeon + 3 MICs a répondu aux exigences de temps de calcul du LTE avec un temps maximal près de 50 µs et un temps moyen de 31 µs.

Enfin, nous avons utilisé le travail précèdent pour paralléliser l'exécution d'une FFT sur les plateformes Xeon et Xeon-Phi (MIC). Cette implémentation parallèle d'une FFT consiste à diviser des données en sous-parties, exécuter l'algorithme des FFTs indépendantes pour chaque sous-partie et recombiner les résultats de notre FFTs indépendantes pour avoir un résultat final. En dehors des normes LTE, nous avons aussi étudié le cas de longues FFT (N). Les résultats montrent une vitesse de calcul dépassant la méthode de référence bien connue pour sa vitesse, la FFTW, pour une longueur de FFT

$N > 2^{18}$ et aussi MKL pour $N > 2^{19}$. Ce gain en vitesse peut atteindre un gain de 4 fois plus rapide sur Xeon et de 2.5 sur Xeon-Phi pour $N=2^{29}$ par rapport à FFTW sur 16 threads. En comparaison à la FFTW sur un seul thread, un gain en vitesse de 20 et 25 fois plus rapide sur Xeon-Phi (MIC) et sur Xeon, respectivement.

Remerciement

Je voudrais remercier mes parents Ammar et Wided et mes sœurs Afef et Wafa pour leur aide, soutien et assistance durant toutes les étapes de ce modeste travail. Je tiens à remercier mes amis Ghassen et Hocine pour leurs précieux conseils, mon directeur de recherche Daniel Massicotte, mon codirecteur Yvon Savaria et tous ceux qui ont contribué à réaliser ce projet. Je tiens aussi à remercier la Société Canadienne de Microélectronique (CMC) et Calcul Québec pour leurs infrastructures, leurs formations spécialisées et leurs services de centre de calcul. Finalement, je tiens à remercier le Regroupement Stratégique en Microsystèmes du Québec (ReSMiQ) pour le soutien financier.

Table des matières

Liste des tableaux

Liste des figures

Liste des symboles

AMPS	Advanced Mobile Phone Systems
API	Application Programmable Interface
CDMA	Code Division Multiple Access
CPU	Central Processing Unit
C-RAN	Cloud and Centralized Radio Access Network
DPDK	Intel Data Plane Development Kit
FDMA	Frequency Division Multiple Access
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
GPRS	General Packet Radio Service
GPU	Graphic Processing Unit
GPGPU	General Purpose Graphic Processing Unit
GSM	Global System for Mobile network
HARQ	Hybrid Automatic Response reQuest
HPC	High Performance Computing
HSPA	High Speed Packet Access
IEEE	Institute of Electrical and Electronics Engineers
LTE	Long Term Evolution
MIMO	Multiple Input Multiple Output

MKL	Intel Math Kernel Library
MIC	Many Integrated Core
MPI	Message Passing Interface
OFDM	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
OpenMP	Open Multiprocessing
PMFFT	*P*arallel multithreaded FFTs
QPSK	Quadrature Phase Shift Keying
QAM	Quadrature Amplitude Modulation
SMFFT	Sequential Multithreaded FFT
SMS	Short Message Service
TDMA	Time division Multiple Access
TACS	Total Access Communication System
UMTS	Universal Mobile Telecommunications System
W-CDMA	Wideband - Code Division Multiple Access
WIMAX	Worldwide Interoperability for Microwave Access

Chapitre 1 - Introduction

LTE *(Long Term Evolution)* présente une technologie émergente et prometteuse pour fournir un accès rapide à l'Internet qui peut atteindre 100 Mbit/s pour une bande passante de 20 MHz. Dans la couche physique de la LTE, le flux de données est parallélisé et réparti sur des sous-porteuses pour la transmission. Le procédé de modulation des symboles de données est basé sur OFDM (Orthogonal Frequency Division Multiplexing). L'orthogonalité est assurée à l'aide de la transformée de Fourier inverse (IFFT) à l'émetteur et la transformée de Fourier directe (FFT) au niveau du récepteur [1]. La couche Physique de LTE est définit comme suit (Figure 1-1).

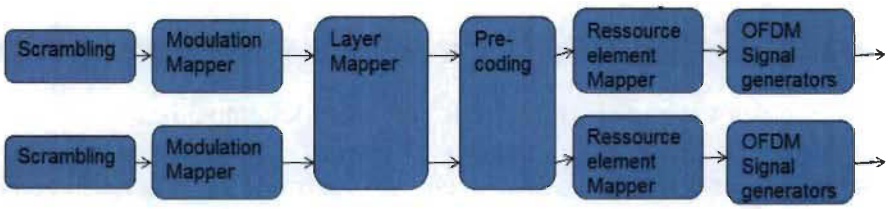


Figure 1.1 La couche physique du LTE [3]

Scrambling : Dans les systèmes de télécommunication, un scrambler est un dispositif qui transpose les signaux et facilite la communication entre l'émetteur et le récepteur.

Modulation Mapper : La modulation transforme les valeurs des bits d'entrée à des symboles de modulation complexes suivant le type de modulation spécifié ; QPSK, QAM16 ou QAM64.

Layer Mapper : Il divise la séquence de données obtenue après la modulation dans le nombre des couches disponibles.

Précodage : Il est utilisé pour la transmission et la communication sans fil multi antennes. Le même signal est émis par chacune des antennes avec une pondération appropriée (phase et gain) de telle sorte que la puissance du signal est maximisée à la sortie de l'émetteur.

Ressource Element Mapper : Il sert à mapper les quadruplets des symboles OFDM dans une sous-trame.

OFDM : C'est un procédé de codage de données numériques sur plusieurs porteuses pour une transmission rapide des données.

Le principal problème dans la couche physique du LTE, plus précisément dans le bloc "Resource Element Mapper", est l'exécution des 100 FFTs indépendantes d'après le protocole LTE. Pour une bande passante de 20 MHz, la longueur de la FFT est de 2048 points d'après la version 8 de la norme LTE [3]. Durant l'exécution séquentielle de la FFT, nous avons remarqué des valeurs aberrantes qui dépassent le temps d'exécution limite des FFTs. Ce dépassement en temps de calcul risque de mettre la fiabilité de la technologie LTE en question [1]. Considérant que la diminution des coûts de calcul est devenue un exemple typique et clair dans la science informatique, ce nouvel exemple fait preuve que nous entrons dans une nouvelle ère où la vitesse et la performance en Téra flops sont devenues une clé dans plusieurs champs de recherche.

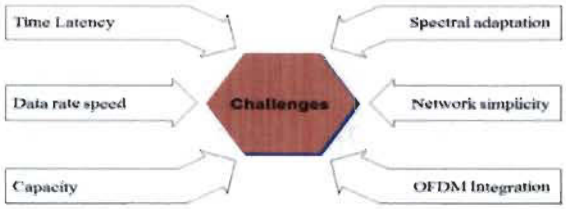


Figure 1.2 Les défis de LTE

1.1 Problématique

Le protocole de la LTE est déjà défini dans la version 8 [5]. Donc, avec tout ce que nous allons changer dans la conception de canal, nous devons respecter les exigences et la capacité du nombre d'utilisateurs simultanés pour que le système accepte un nombre très élevé d'utilisateurs par cellule [2]. Il est aussi indispensable de conserver l'efficacité spectrale pour assurer une bonne capacité du système LTE [2] et la vitesse pour atteindre 100 Mbit/s pour le "*download*". L'une de nos principales problématiques est de réduire la latence du système et l'amélioration de sa capacité avec les différentes bandes passantes de LTE ; 1.4 MHz, 3 MHz, 5 MHz, 10 MHz, 15 MHz et 20 MHz à la fois avec le *download* et le *upload*. Lorsque l'utilisateur mobile est en mouvement ; sa fréquence change avec sa position, c'est pour cette raison qu'il faut assurer la robustesse du système.

Le défi est de transférer la couche physique (*baseband processing*) actuellement située à la base du BTS (Base Tranceiver Station) vers les Nuages. Ceci afin de centraliser le matériel et le calcul des couches physiques des BTS. Au niveau architectural, un nouveau concept issu des technologies liées aux centres de traitement de données, appelé bande de base centralisée des réseaux d'accès radio (C-RAN – *Cloud and Centralized Radio Access Network*), émerge et semble prometteur. L'avantage du C-RAN réside dans sa capacité à mettre en œuvre les fonctionnalités LTE avec une très faible latence entre plusieurs

fonctions radio. Il utilise des plateformes et des technologies de virtualisation en temps réel enracinés dans le nuage pour atteindre l'allocation dynamique de ressources partagées [41] [42] [43]. Ceci est assuré avec les systèmes Multicœur et Manycoeur.

La mise en œuvre n'est pas une tâche facile à faire. Pour implémenter des codes sources sur des processeurs ou des coprocesseurs, nous avons rencontré des nombreux défis.

Concernant les défis matériels dans ce mémoire, ils se résument dans quelques points :

SMP (Xeons) Versus Cluster (MICs) : Le système SMP est un matériel à plusieurs processeurs de type Xeon (CPU) qui possède deux ou plusieurs processeurs de même type. Ces systèmes partagent la même mémoire principale commandée par le système d'exploitation. Ils utilisent l'architecture symétrique pour traiter les données. Dans l'architecture SMP, la mémoire est partagée. Il est donc très facile à utiliser plusieurs cœurs, ce qui conduit à un très haut parallélisme. La plupart des programmes parallèles sont basés sur cette architecture. Par contre, le cluster possède une vitesse de connexion plus lente par rapport SMP. Il peut fournir un plus grand nombre de processus parallèles, ainsi que d'autres ressources matérielles comme la mémoire distribuée entre les nœuds. À cause de non-partage du mémoire, le cluster doit utiliser des outils de transfert de messages, comme MPI, pour assurer le parallélisme au niveau des processus [38].

- Dans ce mémoire, nous devons garantir le parallélisme des 100 threads et il reste lié à la plateforme Multithreaded. Pour soutenir la plateforme Multithreaded, le coprocesseur a des caractéristiques telles que l'architecture de base, le flux des données et la mémoire cache intérieure. D'une manière générale, chaque noyau a quatre threads et ces quatre threads doivent travailler ensemble pour avoir la meilleure performance générée par ce cœur. L'architecture de ces quatre threads

possède plusieurs composants complexes. Certaines des microstructures sont également incluses, telles que le cache de recharge, le pointeur d'instruction, les drapeaux et le traitement d'unité logique.

- La différence entre le Xeon (CPU) et le Xeon-Phi (MIC) : Le MIC a seulement des dizaines de cœurs. Mais il ne faut pas oublier que les noyaux de MIC sont aussi les processeurs Intel. Intel possède un avantage : l'hyper-threading. Un noyau physique peut être simulé à deux cœurs logiques. Cependant, chaque noyau de MIC peut exécuter quatre threads en même temps. Ils sont différents des hyper-threads sur un CPU puisque la performance est considérablement augmentée, et chaque thread peut être traité comme un véritable noyau [32].
- Dans chaque programme, nous avons besoin de réduire les latences de mémoire pour chaque thread. Chaque noyau de calcul peut exécuter deux instructions dans chaque cycle d'horloge. La synchronisation entre les pipelines et les diffusions est contrôlée par le système interne pour aider l'accélération d'exécution des données.
- Nous devons accéder aux données locales des noyaux pour contrôler le parallélisme. Le coprocesseur étend un pipeline amélioré de 64 bits. Le noyau de MIC inclut le Multithreading, qui affecte les performances, la latence et l'unité d'exécution des données.

Pour les problèmes logiciels que nous avons rencontrés dans ce mémoire :

Le contrôle des threads est lié nécessairement au mode d'accès de la mémoire cache. Chaque noyau est indépendant l'un de l'autre, avec un cache de 512 ko. Les noyaux sont reliés entre eux par un bus interne à grande vitesse. Lorsque le coprocesseur est en cours

d'exécution d'une tâche, la mémoire cache L2 est en train de faire une diffusion en flux (Streaming) des données avec une certaine vitesse.

- Comment équilibrer la charge entre les noyaux d'une machine ? La variable KMP_AFFINTY dans OpenMP sert à optimiser l'équilibrage de charge entre les cœurs. Le meilleur équilibrage peut être atteint s'il est mis en place correctement.
- Sur Xeon-Phi, nous avons besoin d'avoir une bonne pratique des deux modes d'exécution : *Native* et *Offload.* Le mode *Offload* est généralement utilisé dans la structure à un seul nœud en utilisant OpenMP. Il est basé sur le CPU, en tenant compte du côté MIC comme coprocesseur. Le CPU est la partie de la commande, ou l'exécution des tâches de contrôle et les tâches de transmission de données sont faites. Tandis que le mode native est l'exécution complète du code sur la plateforme que ce soit Xeon ou Xeon-Phi.
- Comment détecter le temps perdu causé par les *bugs* de programmation dans la mémoire partagée ? Intel Parallel Inspector XE vérifie les erreurs de Multithreading et l'allocation de mémoire. Inspector XE détecte ces erreurs au moment de l'exécution, qui travaillent habituellement sur une version non optimisée du programme en cours de test. Les courses des threads, les blocages entre eux et leurs emplacements sont aussi détectés.

1.2 Objectifs

Dans le cadre de ce projet, l'objectif principal et à long terme est d'élaborer des stratégies appropriées pour la mise en œuvre de l'OFDM (*Orthogonal Frequency Division Multiplexing*) dans le LTE où l'exécution parallèle des FFTs représente un défi. Ces

stratégies de mise en œuvre tiennent compte du bon choix des caractéristiques de l'émetteur/récepteur (taille de la mémoire, la vitesse du signal d'horloge interne, le nombre des cœurs, l'efficacité spectrale, la mobilité et la capacité du système).

L'objectif principal de ce mémoire est l'implémentation parallèle des FFTs indépendantes sur des plateformes Multicœurs. Pour atteindre cet objectif, nous proposons les sous objectifs suivants : (i) l'implémentation parallèle des FFTs indépendantes avec DPDK, (ii) l'implémentation parallèle des FFTs indépendantes avec OpenMP, (iii) l'implémentation parallèle des FFTs indépendantes avec MPI et (iv) la parallélisation d'une FFT.

L'implémentation parallèle des FFTs indépendantes avec DPDK

Cette partie contient l'utilisation des services du Kernel Linux afin d'accélérer l'exécution des FFTs en même temps. Vu que l'implémentation a été faite sur un processeur Core-i7 avec 8 threads, alors la taille maximale de nombre des FFTs parallèles était huit. Notre but était de réduire le temps de latence d'une seule FFT. On a utilisé l'ordonnanceur du Kernel afin d'exécuter 8 FFTs indépendantes. Par contre, on a pas pu utiliser le DPDK sur le serveur de Calcul Québec, car l'exécution des fichiers systèmes du noyau Linux est associé seulement à l'administrateur de serveur et n'est pas dédié aux simples utilisateurs.

L'implémentation parallèle des FFTs indépendantes avec OpenMP

Cette nouvelle bibliothèque nous a permis d'utiliser les serveurs et les outils de Calcul Canada comme OpenMP, le processeur Xeon, le coprocesseur Xeon-Phi, et des autres outils pour la compilation des programmes, débogage et vérification de parallélisme dans le

code. Grâce à cette méthode, on a obtenu des meilleurs résultats par rapport la première. Cela nous a permis aussi d'augmenter le nombre des FFTs parallèles de huit à 100 FFTs et même 200 pour visualiser et analyser les réponses du MIC en cas de surcharge et des situations extrêmes avec des plus grandes FFTs. L'objectif principal dans cette partie est d'analyser les résultats et d'optimiser le code si possible pour atteindre la latence requise en utilisant OpenMP.

L'implémentation parallèle des FFTs indépendantes avec MPI

Les différents outils et la disponibilité des plateformes chez Calcul Québec nous ont donné l'opportunité de regarder plus loin que l'utilisation unique de OpenMP sur un seul MIC. Un nouvel objectif sert à comparer sept types d'implémentations des FFTs. Des solutions logicielles ont donné l'accès aux hybridations entre OpenMP et MPI sur un ou plusieurs MICs. Des plateformes mixtes et hétérogènes étaient un champ d'analyse des résultats afin de mélanger deux types de mémoires ; partagée et distribuée. Notre objectif a été atteint dans cette partie de travail en respectant le temps critique de la norme de LTE. Pour 100 FFTs simultanés avec une largeur de bande de 20 MHz, l'objectif principal est d'exécuter ces FFTs dans un temps inférieur de 67,7 microsecondes [5].

La parallélisation d'une FFT

Les précédents sous-objectifs nous permettront de développer des procédures d'implémentation qui seront utilisées pour l'implémentation parallèle de la FFT, contrairement aux sous-objectifs précédents. Ce sous-objectif utilisera le principe de diviser pour régner, autrement dit il s'agit de diviser une grande FFT au sous-FFTs et d'exécuter les sous-FFTs en parallèle (comme les précédents sous-objectifs) et de recombiner les sous-

FFT pour obtenir la FFT complètes. Notre implémentation sera faite sur Xeon et Xeon-Phi et les résultats seront comparés avec GPGPU.

1.1 Méthodologie

La méthodologie consiste à décrire les stratégies d'implémentation pour OFDM dans le C-RAN. Suite à l'étude des blocs de fonctionnement de l'OFDM, les blocs suivants ont été notés comme plus complexes que les autres : Codeur/Décodeurs, FFT, inversion de matrices et estimation de canal. Le codeur/décodeur montrant la plus grande complexité où des solutions ont été proposées [3][4][5] et à elle seule représente une complexité allant au-delà de notre sujet de maîtrise. Nous avons donc préconisé le reste des calculs de l'OFDM. Compte tenu des expertises sur la FFT du Laboratoire des signaux et systèmes intégrés (LSSI) et des membres du Groupe de recherche en électronique industrielle, nous avons donc entrepris de nous concentrer sur l'exécution des FFTs indépendantes sur les différentes plateformes.

Nous avons utilisé trois plateformes pour l'implémentation dans ce travail. La première plateforme est l'Intel i7-4770, avec une fréquence de 3,6 GHz, 16 Go de RAM DDR4, 20 Mo L3 de mémoire cache, 4 cœurs et 8 threads avec hyper-threading. Le système d'exploitation est Ubuntu 13,04 avec une version du noyau 3.8.8. La deuxième plateforme est Intel Xeon E5-2650, avec une fréquence de 2,0 GHz, 64 Go DDR4 de RAM, 20 Mo L3 de mémoire cache, 8 cœurs et 16 threads. Le système d'exploitation est CentOS 6.5 avec une version du noyau 2.6.3. La troisième plateforme, de Calcul Québec, est le coprocesseur Xeon-Phi (*Many-core*) qui contient 61 cœurs et 244 threads. Chaque noyau fonctionne avec une fréquence de 1 GHz et une taille de 512 Ko de mémoire cache L2 qui interagit avec les autres mémoires caches des autres cœurs. Les noyaux sont reliés entre eux par un anneau

bidirectionnel à haute vitesse. Le coprocesseur (esclave) se connecte avec le processeur Xeon (Maitre) en utilisant le bus PCI pour échanger des données [4]. Nous avons deux coprocesseurs par hôte liée à l'unité centrale de traitement.

Le langage de programmation est C/C ++ et les principaux compilateurs utilisés sont Intel Icc et Gcc. Les bibliothèques MKL, FFTW, MPI et OpenMP sont situées dans le parallel studio XE qui est accessible par le serveur du Calcul Québec de l'Université McGill; Guillimin.

L'exécution des FFTs simultanées est basée sur une étude statistique de 10^6 FFTs sur les plateformes Intel Core i7-4770 et Xeon Sandy-Bridge et 2030 FFTs sur le Xeon-Phi en raison de la capacité de stockage limitée sur le dispositif de serveur.

La validation des résultats de nos implémentations se réalise par l'exécution répétée des FFT sur des structures parallèles. Nous avons mesuré les temps de calcul pour chaque FFT et avons analysé les caractéristiques statistiques de ces temps de calcul. Plusieurs observations ont permis de faire des choix dans nos stratégies d'implémentation et de présenter les meilleures stratégies permettant de satisfaire la contrainte principale, la latence de 67,7 µs pour exécuter 100 FFTs.

La validation des résultats de nos implémentations se réalise par l'exécution répétée des FFTs sur des structures parallèles. Nous avons mesuré les temps de calcul pour chaque FFT et avons analysé les caractéristiques statistiques de ces temps de calcul. Plusieurs observations ont permis de faire des choix dans nos stratégies d'implémentation et de présenter la meilleure stratégie permettant de satisfaire la contrainte principale, la latence. Les tâches ont été exécutées dans les modes séquentiels et parallèles. Dans chaque expérience, les FFTs du nombre flottant complexe et de dimension N sont exécutées sur les

trois plateformes. Pour analyser la distribution de 10^6 temps de calcul, nous avons examiné les mesures statistiques pour démontrer la performance de la méthode basée sur trois caractéristiques clés [12] : la tendance centrale, la dispersion et la forme. Les moments de distributions sont à définir par le premier moment (moyenne), le maximum, le coefficient de variation (CV) et les troisième et quatrième moments (dissymétrie et kurtosis). Le dernier donne plus d'informations pour mesurer les valeurs aberrantes de distribution sujettes autour de la moyenne et d'atteindre un temps de calcul déterministe. Pour 100 FFTs, nous avons besoin de ne pas dépasser le maximum de 67,7 µs.

1.2 Organisation du mémoire

Dans le chapitre 2, on va aborder l'importance du 4G LTE et comment son évolution est de plus en plus importante dans les réseaux de télécommunication. Ensuite, les outils logiciels et les techniques utilisées sont mentionnés.

Dans le chapitre 3, une étude sur l'utilité de DPDK et le noyau Linux et leur rôle dans l'accélération des processus de calcul ont été établis. L'algorithme de partage des tâches sur plusieurs threads est expliqué pour bien démontrer l'aspect Multithread et parallèle.

Dans le chapitre 4, une étude de notre modèle parallèle avec OpenMP sur les trois plateformes (Intel Core-i7, Xeon et Xeon-Phi) est introduite. Les techniques d'affinité des threads sont expliquées et implémentées sur les plateformes. Une comparaison entre les plateformes et les types de mémoire utilisés (partagée ou distribuée) est présentée. Les deux types d'implémentation sur Xeon-Phi (Native et Offload) sont mis en évidence.

Dans le chapitre 5, sept implémentations des FFTs natifs sont comparées. Les environnements d'exécution considérés sont : OpenMP (Open Multi-Processing) sur 1 MIC,

MPI (Message Passing Interface) sur 1 MIC, 2 MICS, et 3 MICS, OpenMP + MPI sur 1xMIC et 1xMIC + 1xXeon et MPI sur une plateforme hétérogène composée de 3xMICs + Xeon. Une comparaison entre le degré du parallélisme des outils logiciels OpenMP et MPI est introduite. En outre, la nouvelle hybridation OpenMP + MPI est expliquée.

Dans le chapitre 6, on va expliquer la FFT parallèle avec des larges entrées pour les systèmes à haute performance. En passant par les trois phases de développement ; dispersion, traitement et recombinaison. On va aussi donner des solutions pour réduire le temps d'accès au mémoire vu l'immense nombre des données afin de faciliter l'algorithme parallèle et assurer son optimisation.

Le dernier chapitre est une conclusion générale de ce travail suivi d'une bibliographie.

Chapitre 2 - LTE et les FFTs indépendantes pour les systèmes Multicœur/Manycœur

2.1 LTE (Long Term Evolution)

Historique

Tout a commencé avec les réseaux mobiles de première génération tels que les AMPS aux Etats-Unis, le TACS au Royaume-Uni et la Radiocom en France, ces technologies étaient les meilleures parmi leur genre au cours des années 80. Elles étaient basées sur les accès multiples en fournissant à chaque utilisateur une fréquence, mais leur capacité reste limitée avec un nombre d'utilisateurs qui a atteint seulement 60 000 utilisateurs. Toute la conception était analogique. En outre, il y avait un fardeau de transmission lorsque deux appels viennent à l'utilisateur mobile au même temps, les bornes n'ont pas été optimisées avec un équipement grand et lent. Ils n'ont pas duré longtemps [2] [3] [4].

Dans les années 90, le GSM en Europe est venu avec une grande amélioration. Grâce à l'évolution du traitement des signaux numériques et les tâches hyperfréquences, les équipements ont été réduits en taille et ils deviennent plus modernisés en permettant une grande mobilité pour l'utilisateur [2] [4].

Technologiquement parlant, les technologies 2G étaient basées sur la modulation numérique comme FDMA et TDMA, en utilisant le temps et la fréquence de multiplexage. À la fin des années 90, la station de base pour le réseau GSM est devenue connectée avec le GPRS pour améliorer la vitesse qui a atteint 240 kbit/s. Le GSM était un succès dans le

monde entier et a duré longtemps, mais sa capacité système était limitée. Ce qu'était une raison pour la migration vers les technologies 3G. Prendre le GSM comme un exemple réussi, l'UMTS est né comme une nouvelle fixation du GSM et la technique de codage du CDMA. Cet UMTS représente la nouvelle ère de la 3e génération dans les systèmes de communication mobiles [2] [4].

La 3G a commencé avec l'UMTS, CDMA a évolué pour W-CDMA avec une porteuse occupant un canal de 5 MHz et une bande passante de 3,8 MHz [2]. L'objectif était de réduire la taille des terminaux et des systèmes mobiles. La vitesse UMTS était de 380 kbit/s au lieu de 118 kbit/s du GSM/GPRS. Donc, pour améliorer cette vitesse, il y avait une migration vers une nouvelle technologie appelée HSPA. Cette technologie a utilisé une nouvelle technique de modulation avec le nom de la QAM16 pour le "downlink" et QPSK pour le "uplink". Le mélange de ces nouvelles façons a donné naissance au HARQ pour la transmission entre l'émetteur et le récepteur afin de réduire la latence des paquets transmis entre les deux parties. Ce HSPA offre à l'utilisateur une vitesse de "download" de 14 Mbit/s et une vitesse de "upload" de 5 Mbit/s.

Plus tard, une nouvelle version de HSPA a été libérée avec le nom de HSPA+ qui a utilisé les mêmes techniques de codage que le HSPA, mais avec une modulation améliorée qui est le QAM64. En conséquence, le temps de latence a été réduit à 30ms, et la vitesse de *download* a atteint les 42 Mbit/s [2] [3].

Tableau 1 Comparaison entre GSM/UMTS/HSPA/HSPA+ [2]

	GSM/GPRS	**UMTS**	**HSPA**	**HSPA+**
Vitesse Max pour le *upload*	118 Kbit/s	384 Kbit/s	5.8 Mbit/s	11.5 Mbit/s
Vitesse Max pour le *download*	236 Kbit/s	384 Kbit/s	14.4 Mbit/s	42 Mbit/s
Latence	300 ms	250 ms	70 ms	30 ms
Largeur de bande de canal	200 Khz	5 MHz	5 MHz	5 MHz avec 2 canaux en même temps
Modulation pour *download*	GMSK	QPSK	QPSK, 16QAM	QPSK, 16QAM, 64QAM
Modulation pour *upload*	8PSK	BPSK	BPSK, QPSK	BPSK, QPSK, 16QAM
Fréquence de la bande (MHz)	900/1800	900/2100	900/2100	900/2100
Technique d'accès	FDMA/ TDMA	CDMA	CDMA/ TDMA	CDMA/ TDMA

L'UMTS et HSPA+ sont largement utilisés dans plusieurs usages tels que (Internet, application mobile, TV ...) grâce à l'apparition des nouveaux terminaux comme (téléphones intelligents, 3G + clés, mini-ordinateurs ...) [2] [3].

Contexte industriel

Le 3rd *Generation Partnership Project* (3GPP) est composé de nombreuses sociétés portant le nom de "développeurs du GSM" au cours des 20 dernières années. Le WIMAX a été basé sur l'OFDM avec une meilleure capacité que la version précédente de l'UMTS et son évolution le HSPA [2].

Vers le LTE (Long Term Evolution)

Depuis 2006, une nouvelle révolution a été faite qui a mis le LTE comme le grand titre de la 4G. Une nouvelle génération a commencé avec un nouveau défi qui nous a conduits à une petite latence et une meilleure vitesse par rapport la technologie HSPA+ [2][5].

LTE Vs WiMAX

Les méthodes de télécommunication s'améliorent et beaucoup de nouvelles technologies envahissent le monde comme le WiMAX et le LTE. Pourquoi le LTE est-il plus efficace que le WiMAX [6] [7] ? Parlant de performance, LTE atteint le 100Mb/s pour la vitesse de téléchargement, avec 25Mb/s pour le WiMAX. Le LTE est intégré dans la nouvelle technologie de téléphonie mobile et il est compatible avec les appareils cellulaires récemment conçus dans le monde, ce qui en fait le meilleur sprint grâce à ses nouvelles techniques de traitement de codage et les canaux rapides avec le temps d'exécution dans l'ordre des microsecondes [6]. LTE a un avantage caractérisé par sa résistance au bruit dans les signaux. Par contre, WiMAX ne fonctionne pas bien avec les appareils 4G et trouve du mal à s'adapter avec les téléphones 3G [1].

LTE (Long Term Evolution)

LTE représente une technologie émergente et prometteuse pour fournir un accès Internet à large bande omniprésente. Pour cette raison, plusieurs groupes de recherche tentent d'optimiser ses performances [8].

Elle met en place des débits de données efficaces dans le processus de transmission, une largeur de bande flexible qui peut être commandée à partir de 1.4 MHz à 20 MHz et à faible temps de latence pour la couche physique [9]. En plus des autres blocs de la couche,

tels que le codage de canal, l'ordonnancement, l'adaptation de liaison, le mapping et l'optimisation de canal. Le plan de transmission en liaison descendante est construit sur l'OFDMA qui transforme le canal de fréquence disponible dans une collection de plusieurs sous-canaux ; chaque sous-canal est basé sur la sous-porteuse représentant un utilisateur [6] [1].

Les sous-canaux profitent de la mise en œuvre du MIMO contrairement à la transmission WCDMA [9]. L'utilisation de l'OFDMA se fait essentiellement dans le domaine des fréquences pour rendre le traitement plus facile et la programmation moins compliquée. Ces sous-canaux seront filtrés afin d'avoir un bon gain pour les utilisateurs [5].

Dans les systèmes de télécommunications, les données améliorées sont souvent structurées autour des récepteurs/émetteurs radio. Ces récepteurs/émetteurs travaillent généralement dans les réseaux cellulaires où le traitement de signal numérique est en constante évolution afin d'améliorer la performance. L'augmentation massive des émetteurs/récepteurs ouvre des nouvelles voies d'implémentation de la couche physique sur des plateformes Multicœurs. Ces plateformes offrent principalement la flexibilité opérationnelle pour une meilleure fiabilité et l'efficacité des méthodes clés. Ainsi la modélisation du système embarqué doit passer par une étude détaillée de l'architecture et une bonne modélisation de l'émetteur/récepteur LTE [9] [5].

Les caractéristiques du LTE

Il est important de considérer la capacité des systèmes LTE, la vitesse de débit de données et la latence qui sont les défis les plus importants que nous rencontrons au cours de ce mémoire, en commençant par l'étude de la FFT et en terminant par l'implémentation.

Capacité : En expliquant la différence entre la capacité et la vitesse. La capacité est le nombre maximal d'utilisateurs par cellule qui sont en mesure de se connecter simultanément lorsque le réseau est entièrement chargé. Pour avoir une bonne capacité, nous devons avoir une bonne efficacité spectrale. Le problème reste lorsque plusieurs utilisateurs se connectent au réseau en même temps, donc la vitesse (Mbit/s) vient à être réduite et partagée entre tous les autres utilisateurs. Par conséquent, la vitesse pour un seul utilisateur peut être affectée à l'efficacité spectrale, divisée en nombre d'utilisateurs actifs. La capacité d'un réseau est liée au nombre d'utilisateurs actifs simultanément [2].

Les besoins accrus pour la capacité sont dans une évolution stable avec les réseaux mobiles. En effet, les progrès technologiques des réseaux encouragent l'utilisation de nouveaux types de méthodes pour assurer une expérience plus confortable à utilisateur. L'utilisation intensive du réseau et le grand nombre d'utilisateurs font le problème de la capacité d'une énigme qui doit être résolu. Les exigences de capacité sont en croissance et la technologie doit évoluer [2].

Le gain de capacité pour le HSPA et HSPA+ sont améliorées si l'on compare au GSM et l'UMTS. Mais même avec cette amélioration, il ne répond toujours pas aux exigences du nombre croissant d'utilisateurs par jour. Grâce à la nouvelle progression de la technologie et de l'énorme diffusion des téléphones intelligents et les mobiles 4G+, le nombre d'utilisateurs est étonnamment en augmentation qui conduit à une augmentation du nombre d'utilisateurs et l'élargissement des capacités est devenu obligatoire. Le facteur de croissance annuelle du nombre d'utilisateurs dans le monde a dépassé les 100% en 2011 et ce taux devrait se poursuivre dans les années suivantes. La solution est d'ajouter d'autres sous-porteuses dans la couche physique de LTE. Cela n'est pas facile, car l'activation de

plusieurs sous-porteuses est liée à la taille de la largeur de bande qui doit être respectée. Dans de nombreux pays, le nombre des sous-porteuses est limité et comme résultats, on peut remarquer un rejet des appels dans les heures de pointe.

Vitesse : Une vitesse plus élevée que celle fournie par le réseau HSPA est nécessaire en raison de la forte demande des utilisateurs et des opérateurs. Cette exigence est d'abord motivée par le désir d'offrir la mobilité à l'utilisateur comparable aux réseaux résidentiels. La vitesse fait la différence entre les opérateurs et les pays, c'est pourquoi le déploiement de LTE est venu de prouver une meilleure confiance de vitesse [2].

Temps de latence : La latence en LTE est le retard généré et causé par le système. Il existe deux types de latence : Latence de commande et latence de l'utilisateur. La latence de commande est le temps nécessaire pour la connexion et l'accès au réseau. Le temps de latence de l'utilisateur représente le temps nécessaire pour transmettre les paquets de données juste après la connexion. En général, le temps de latence donne une indication sur la capacité du système à traiter toutes les données à venir et de transmission. Par exemple, le HSPA offre un temps de latence de 70ms qui permet l'utilisation du service de jeux vidéo en ligne. La latence améliorée avec le LTE est l'un des objectifs les plus importants dans le projet LTE [2].

Vers la communication OFDM

La conception de la théorie a été inventée dans les années 80 et a commencé depuis 2000. Finalement, l'adaptation de cette technique dans le terminal est devenue possible grâce à l'évolution du traitement des signaux numériques et les nouvelles améliorations d'équipements micro-ondes. L'OFDM offre de nombreux avantages pour le Nouveau Monde de communication mobile ; il convient de mentionner son immunité aux symboles

d'interférence avec les objets et les bâtiments. La partie motivante avec l'OFDM est qu'il est capable de manipuler des bandes passantes dans le même temps, de nombreux utilisateurs différents avec des vitesses de données différentes. Le défi a été accepté d'intégrer l'OFDM avec le LTE et de migrer de l'ancien CDMA à l'OFDMA [2].

La simplicité du réseau

Le réglage du réseau mobile est très couteux pour les opérateurs. Ça implique d'abord le déploiement de stations de base. Il exige aussi des paramètres de configuration initiale de l'équipement installé. Ces tâches de configuration sont fastidieuses, et peuvent causer des erreurs qui dégradent la qualité du service offerte aux utilisateurs. Par exemple, le GSM est basé sur des nombreux composants analogiques qui causent des longs temps de latence dans la communication entre les nœuds. En outre, les opérateurs ont optimisé ces paramètres afin de maximiser la qualité de services et de la capacité du système [2].

2.2 Les FFTs indépendantes et la FFT parallèle

Pour les FFTs indépendantes avec DPDK, le noyau Linux et les techniques DPDK trouvés dans la littérature "*the network stack latency for game servers*" [10] étudient l'effet du temps moyen et son écart-type sur la performance du système. Leurs mesures ont montré que la taille de mémoire tampon a une influence sur le retard. Une taille faible a conduit à un problème d'optimisation et débit. Avec "*the fast user space packet processing*" [11], l'implémentation a montré que l'utilisation de DPDK a aidé à gagner, jusqu'à 2,3 fois en gain de vitesse par rapport à d'autres implémentations logicielles. L'implémentation dans "*DPDK implementation of various applications using networks on end user nodes*" a montré que les expériences peuvent atteindre des vitesses de transmission

des paquets 9 fois supérieure à celle d'une implémentation basée sur les sockets [12]. Dans les systèmes d'exploitation Linux, l'application de la FFT est basée sur des serveurs Linux à l'aide des processeurs Intel Xeon E5506. Les expériences indiquent que le composant avec des threads peut être utilisé pour les applications à hautes performances [13] [17]. Un algorithme radix-2 de la FFT est développé pour les architectures SIMD (Single Instruction-Multiple Data) génère un temps de calcul moyen de 69 µs sur Intel Pentium 4 [14]. Notre contribution utilise toutes les techniques suivantes pour accélérer la mise en œuvre de la FFT en utilisant DPDK avec FFTW et MKL sur la plateforme Intel Core i7. Cela comprend différents scénarios de taille de la mémoire, la configuration de la bande passante et de l'isolation des cœurs sur Linux. Cela devrait conduire à une connaissance précieuse sur le temps de traitement, la consommation d'énergie, la complexité du matériel, et l'efficacité du logiciel utilisé.

Pour les FFTs indépendantes avec OpenMP, les techniques de parallélisation mis en évidence dans la littérature examinent l'implémentation de huit FFTs parallèles appropriées pour la OFDM en utilisant huit FFTs parallèles fonctionnant dans des longueurs variables de 64/128/256/512 [15]. En outre, les auteurs montrent comment l'augmentation du nombre de threads et de la taille des données d'entrée influence la performance avec OpenMP [16]. L'OFDM souffre d'un temps de traitement élevé dans la couche physique et la latence est une contrainte critique pour contester l'exécution des FFTs indépendantes. Les processeurs Multicœurs sont convenables pour les réseaux radio centralisés (C-RAN) et offrent une plateforme plus configurable que la FPGA et l'ASIC [18] [24]. Le LTE peut adopter le Xeon-Phi comme un substitut au FPGA et DSP. Ces derniers nécessitent une programmation complexe de bas niveau à la différence d'une simple programmation

standard en C/C ++ pour le Xeon-Phi. Notre contribution utilise toutes les techniques suivantes pour mettre en œuvre 100 FFTs indépendants sur Xeon-Phi pour la couche physique de LTE. En utilisant les modes Native et Offload pour l'implémentation sur Xeon-Phi, un gain de temps de calcul pourrait être obtenu pour notre code parallèle. De plus, nous allons comparer les résultats avec Xeon. Cela devrait conduire à des connaissances précieuses sur le temps de traitement et l'efficacité de chaque plateforme.

Tableau 2 LTE Uplink physical layer parameters

Largeur de canal (MHz)	1.25	2.5	5	10	15	20
Taille de FFT	128	256	512	1024	1536	2048
Nombre des RB	6	12	25	50	75	100

Il a été constaté qu'OpenMP peut souffrir de fuites de mémoire et d'un ralentissement du temps d'exécution [16]. On a eu une évolution limitée que lors de l'exécution de 100 FFT. Le temps de traitement peut atteindre un maximum de 180 µs, tandis qu'une valeur moyenne de 120 µs a été observée sur le MIC [19]. L'implémentation avec MPICH2 pour les fibres permet l'exécution de centaines de milliers de processus MPI sans avoir besoin d'un grand nombre des cœurs [20]. Pour confirmer l'efficacité d'hybridation OpenMP+MPI, MPI a été utilisé pour la communication inter nodale externe de la région parallèle et OpenMP a été utilisé à l'intérieur des nœuds [21]. Dans une étude comparative, des taches parallèles ont été implémentées en utilisant : 1) MPI pour la coordination des tâches sur une grande mémoire distribuée sur un SMP, 2) OpenMP sur un système de mémoire distribué au sein de la plateforme Altix 3700 et une mémoire partagée Dell PowerEdge 6650, et 3) une combinaison hybride de MPI et OpenMP mis en œuvre sur les mêmes groupes [22]. La meilleure vitesse a été obtenue avec MPI pure sur une mémoire partagée pour le SMP [22].

Une analyse statistique des performances obtenues lors de l'encapsulation de la transformée de Fourier rapide FFTW dans des environnements de calcul parallèle a été réalisée [23]. Des expériences ont montré que la performance augmente avec le nombre de nœuds [23]. Avec le LTE, la latence est une contrainte critique lors de l'exécution de nombreux FFTs indépendantes. Les techniques et les architectures suivantes sont envisagées : Xeon et Xeon-Phi avec OpenMP, MPI et l'hybridation OpenMP + MPI.

Pour la FFT parallèle, il a été constaté que la méthode d'implémentation de la FFT parallèle 1-D sur un GPGPU est performante. Les résultats montrent que l'implémentation est 3,62 fois plus rapide par rapport à la FFTW sur la plateforme Xeon pour effectuer 64M points d'une FFT1-D [25]. Nous allons comparer nos résultats avec cette référence dans une étape ultérieure. Selon d'autres études, il est évident que des FFTs implémentées sur Xeon et Xeon-Phi sont performantes en utilisant OpenMP [19]. Un projet qui résout la FFT en divisant l'ensemble du problème en plusieurs parties parallèles était un bon soutien à notre travail. Leurs résultats expérimentaux pour la DCT (transformée en cosinus discret) en utilisant les algorithmes gg90 ont montré comment l'algorithme gg90 parallèle fonctionne mieux que l'algorithme de levage séquentiel [26]. Une décomposition hybride et une implémentation parallèle de la méthode Car-Parrinello ont été obtenues en utilisant deux algorithmes ; la décomposition spatiale et orbitale hybride du problème qui a permis d'atteindre des performances très élevées avec le nombre croissant des nœuds [27]. Une efficace FFT 3-D pour les ondes planes sur des machines massivement parallèles a été étudiée. Les résultats ont montré que l'utilisation de leurs nouvelles techniques d'optimisation parallèles a permis d'atteindre des performances très élevées allant jusqu'à 130 GFLOPS sur 256 processeurs dans le cadre d'onde plane à structure électronique [28].

L'article "*Parallel zero-copy algorithms for FFT and conjugate gradient using MPI Datatypes*" est un bon exemple de la décomposition de la FFT avec MPI. Leurs résultats de la FFT parallèle et le Solutioneur CG ont montré des améliorations à un facteur de 3,8 et 18% respectivement sur les plateformes BlueGene/P et les systèmes Jaguar [29]. L'évaluation de la performance des algorithmes FFT parallèles ont montré que le gain en vitesse est supérieur en Radix-2 par rapport au Radix-4 et Split-Radix lorsque l'implémentation est faite sur un grand nombre de cœurs. Les approches de parallélisation sont construites sur le fait que le parallélisme augmente de plus en plus avec le nombre des divisions récursives [30]. Une étude de la FFT parallèle multidimensionnelle a montré qu'il est possible d'obtenir une performance élevée sur une grande variété d'architectures parallèles. Leur exécution a été simulée sur 262144 noyaux du supercalculateur BlueGene/P et a prouvé une certaine flexibilité pour le calcul de FFT multidimensionnelle sur des plateformes massivement parallèles [31]. Avec la programmation HPC sur Intel MIC, le but ultime est de réduire considérablement le temps de développement [32].

Notre contribution utilise toutes les techniques suivantes pour mettre en œuvre la FFT parallèle sur Xeon et Xeon-Phi. De plus, nous allons comparer les résultats avec le CPU Xeon Sandy- Bridge et GPGPU.

2.3 Les outils logiciels pour le parallélisme

2.3.1 Data Plane Development Kit (DPDK)

Dans l'environnement Linux, Intel DPDK joue le rôle d'une interface entre la couche utilisateur et le noyau Linux. Le DPDK produit une collection de bibliothèques nécessaires pour la création d'un plan appelé EAL (Environnement Abstraction Layer). Ce plan

contient les éléments de base de DPDK. Par exemple : Timers, mbuf, MemPool, malloc et les threads. Après la création d'EAL, on a la possibilité d'utiliser les utilités de DPDK dans nos programmes. Les différentes couches de DPDK interfèrent avec MKL et FFTW comme indiqué dans la figure 2-1. Pour la gestion de ce plan, Intel DPDK exécute le modèle conçu à l'aide du CPU. La communication entre les threads se produit grâce aux services des threads.

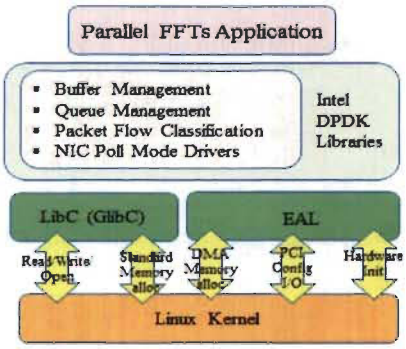


Figure 2.1 Linux Kernel et l'hierarchie DPDK [12]

2.3.2 Math Kernel Library (MKL)

MKL (Math Kernel Library) accélère les routines de calcul mathématiques qui améliorent la performance de la tâche et réduisent le temps de programmation. Intel MKL contient une FFT hautement optimisée et des fonctions d'algèbre linéaire et statistique [33].

L'allocation du Descripteur

Il y a trois fonctions dans cette classe. La première opération est la "*DftiCreateDescriptor*" qui crée un descripteur de DFT (Discret Fourier Transform). Le stockage mémoire est alloué d'une façon dynamique grâce à la routine. Elle met en place le descripteur avec des configurations par défaut selon les valeurs d'entrée souhaitées par l'utilisateur [33]. La seconde est "*DftiCommitDescriptor*" qui fait le précalcul du

descripteur. Cela inclut la factorisation de la longueur d'entrée et le calcul des facteurs. La dernière fonction est "*DftiFreeDescriptor*" qui libère la mémoire allouée par descripteur [33].

Le calcul du DFT

Deux fonctions principales existent dans cette classe. La premiere est "*DftiComputeForward*" pour calculer une FFT directe. La deuxième est "*DftiComputeBackward* " pour calculer une FFT inverse [33].

Configuration du descripteur

Deux fonctions principales existent dans cette classe. "*DftiSetValue*" qui configure une valeur spécifique à l'un des paramètres. "*DftiGetValue*" qui lit la valeur actuelle des paramètres de configuration. Les paramètres sont nombreux, mais ils sont tous traités simultanément [33]. "*DftiCreateDescriptor*" alloue la mémoire pour le descripteur et fait le lien avec tous les paramètres de configuration de données tels que le nom de domaine, la précision, la longueur de transformation et la dimension. Le domaine dans ce cas est celui de transformation directe. La mémoire est allouée de façon dynamique, de sorte que dans la programmation C ++, le résultat est effectivement un pointeur vers le descripteur [33].

Status = DftiCommitDescriptor (FFT_Descriptor);

Après la création du descripteur, nous avons besoin d'une interface qui engage ce descripteur pour effectuer les calculs des FFTs. Dans les plateformes Multicœurs telles que le Xeon, l'implémentation implique l'exploration de plusieurs factorisations de la longueur d'entrée pour avoir une très haute vitesse et une bonne performance.

Status = DftiComputeForward (FFT_Descriptor, buff);

Après la configuration et l'engagement du descripteur, le calcul de DFT aura lieu. Cette transformation utilise le facteur exp (-i2π/N), avec N est la longueur d'entrée stockée dans la variable *buff* [33].

Status=DftiFreeDescriptor(&FFT_Descriptor);

Cette fonction permet de libérer tout l'espace mémoire alloué pour un descripteur. Les actions de l'allocation de mémoire et la libération avec MKL sont commandées par *Intel MKL Memory Manager* [33].

Les instructions suivantes montrent comment utiliser le "*Multithreading*" interne pour Intel MKL avec la FFT. Pour spécifier le nombre de threads avec Intel MKL, nous devons ajouter les paramètres suivants :

EXPORT `MKL_NUM_THREADS = 1` *(MKL avec single threaded mode)*
EXPORT `MKL_NUM_THREADS = 4` *(MKL avec Multithreaded mode)*

Dimension et longueur de transformation

La dimension de la transformée est un entier positif de type MKL_LONG. Pour une transformation 1-D, la longueur d'entrée est un nombre entier positif. Pour la FFT multidimensionnelle, la longueur de chaque dimension est fournie dans un tableau d'entiers [33]. *DFTI_LENGTHS* et *DFTI_DIMENSION* sont les paramètres nécessaires pour la configuration.

Placement des résultats

Par défaut, la routine d'exécution de DFT enregistre les données de sortie dans l'entrée. Dans ce cas, elle remplace la sortie par l'entrée et le paramètre *DFTI_PLACEMENT* prend

DFTI_INPLACE. Nous pouvons utiliser *DFTI_NOT_INPLACE* pour enregistrer les données de la sortie dans une autre variable [33].

Précision des transformations

La configuration *DFTI_PRECISION* présente la précision en virgule flottante dans laquelle la transformation est effectuée. *DFTI_SINGLE* présente la simple précision, et *DFTI_DOUBLE* représente la double précision [33].

2.3.3 Fastest Fourier Transform in the West (FFTW)

FFTW est une bibliothèque pour le calcul des FFTs, développé par Matteo Frigo au MIT (Massachusetts Institute of Technology) [45]. Elle peut calculer des transformations de données réelles et complexes de tailles et dimensions aléatoires en très peu de temps [34].

2.3.4 OpenMP (Open Multi-Processing)

C'est une API qui est conçue aux plateformes avec une mémoire partagée. Elle comprend une collection de directives et routines nécessaires pour un programme parallèle. Le tableau 3 présente les avantages et inconvénients de l'OpenMP.

2.3.5 MPI (Message Passing Interface)

C'est une API de communication utilisée pour les applications parallèles. MPI fournit la capacité de synchronisation et de communication entre un ensemble de processus. Quelques fonctions utiles : Démarrer les processus, envoyer des messages, recevoir des messages et la synchronisation entre les threads. Le tableau 4 présente les avantages et inconvénients du MPI.

Tableau 3 Les avantages et les inconvénients du OpenMP

Les avantages du OpenMP	Les inconvénients du OpenMP
- Elle est très simple grâce à ses routines. Elle n'a pas besoin de gérer le passage des messages comme MPI. - Les directives traitent automatiquement la décomposition des données. - Nous pouvons travailler sur une partie du programme en même temps. - Une très bonne performance sur les plateformes de mémoire partagée telles que Xeon et Intel Core i7, i5, i3. - Des routines flexibles qui ont un gros avantage par rapport MPI. - Elle peut être utilisée sur les GPUs et les MICs pour améliorer les performances.	- Étant donné que la synchronisation se fait automatiquement par la routine, elle peut générer des erreurs et des conditions de course des threads pendant la mise au point. - Elle fonctionne généralement sur les systèmes SMP. - Le type de mémoire est important pour accélérer la performance avec OpenMP. - Des difficultés immenses pour contrôler les discussions des données en cours. - Une grande probabilité de rencontrer un problème de faux partage dans les noyaux.

Tableau 4 Les avantages et les inconvénients du MPI

Les avantages du MPI	Les inconvénients du MPI
- MPI est facile avec les processus sur des plateformes SMP. - Hautement évolutif avec un fort parallélisme. - Contrairement à OpenMP, la synchronisation est la responsabilité du programmeur.	- MPI contient beaucoup des "overheads". - Les tampons de copie nécessitent des transferts importants de données. - Difficile à utiliser dans le codage et la programmation. - Risque de blocage avec la fonction SEND/RECEIVE qui ralentit les performances.

2.4 Environnement et plateformes

Dans ce mémoire, nous avons travaillé sur trois plateformes : Intel i7-4770, Intel Xeon E5-2650 et le coprocesseur Xeon-Phi 7120P. Le tableau 5 explique les caractéristiques des trois plateformes.

Tableau 5 Les plateformes Multicœurs et Manycœurs

Intel i7-4770	**Intel Xeon E5-2650 CPU**	**Xeon-Phi 7120P coprocesseur**
C/C++/Fortran; OpenMP/MPI	C/C++/Fortran; OpenMP/MPI	C/C++/Fortran; OpenMP/MPI
Ubuntu 13.04, Kernel 3.8.8	Standard Linux OS	Special Linux distribution
16 GB DDR4 of RAM, 20MB L3	Up to 768 GB of DDR3 RAM	6-16 GB cache GDDR5 RAM
4 cores, 8 threads	**8 cores, 16 threads**	**61 cores, 244 threads**
3.6 GHz	3 GHz	1 GHz
	2 threads per core	4 threads per core
	256-bit AVX vectors	512-bit vectors

Chapitre 3 - L'implémentation parallèle des FFTs indépendantes avec DPDK

3.1 Introduction

L'énorme évolution du noyau Linux a mis en place ce dernier comme la plus célèbre option pour les appareils mobiles, en concurrence avec d'autres systèmes d'exploitation comme Windows. Dans les systèmes de télécommunication, la transformée de Fourier discrète (DFT) est utilisée dans le traitement du signal numérique afin d'assurer la modulation et la réception/transmission des données sur les téléphones mobiles. La transformée de Fourier rapide (FFT) présente un algorithme important pour calculer la DFT. Pourtant, avec l'augmentation de la taille d'entrée de la FFT, plus de traitement doit être utilisé. La FFT est largement nécessaire pour la technologie industrielle pour concevoir d'autres algorithmes sophistiqués où le temps de calcul joue un rôle important.

Nous présentons l'utilisation du Intel Core i7 pour calculer la FFT de longueur 2048 avec Intel DPDK et les bibliothèques FFT ; MKL et FFTW. À cause du haut temps écoulé et les valeurs aberrantes visibles qui peuvent provoquer une défaillance de la transmission des paquets dans le téléphone mobile, le noyau Linux est une solution pour assurer un calcul robuste de FFT en utilisant les services de base de noyau. Nous avons réussi à effectuer une étude statistique en calculant le temps FFT un million de fois afin d'extraire : Le minimum, maximum, valeur moyenne, variance et la valeur de kurtosis. En outre, nous

mettons en évidence l'efficacité de la DPDK et son influence positive dans l'accélération et la stabilisation de la FFT.

3.1 Le noyau Linux et DPDK

L'objectif de DPDK est de fournir un environnement facile pour un traitement de paquets rapide dans les applications de plan de données. Cet utilitaire crée un ensemble de bibliothèques pour certains environnements grâce à la construction de l'EAL qui est conçu pour les processeurs Intel. Le EAL doit commencer, lancer et allouer des ressources de bas niveau, pour couvrir les particularités des bibliothèques de l'environnement, et de gagner l'accès aux ressources, tels que l'espace mémoire, noyaux, la distribution des threads et l'allocation mémoire qui joue un rôle important dans le développement de notre application [35] [36]. Une fois la bibliothèque EAL est créée, nous pouvons la lier avec les bibliothèques MKL et FFTW. Des autres bibliothèques sont également fournies tels que l'utilitaire ring, tampon et la gestion des fichiers d'attente. Pour installer DPDK sur notre plateforme, nous avons besoin d'un système d'exploitation Linux pour accéder aux différentes routines du noyau. Cependant, les bibliothèques MKL et FFTW sont considérées externes à DPDK et son environnement, un *Makefile* est important pour lier toutes les bibliothèques requises ensemble où nous indiquons les chemins des fichiers .c et .h.

Pour faire une allocation mémoire et d'éviter le temps de transition entre les tampons, le paramétrage du *Hugepage* est très pratique pour augmenter les performances et assurer une exécution à grande vitesse dans la mémoire cache.

3.2 Implémentation et isolation des cœurs

Dans ce travail, nous appelons huit FFTs indépendantes sur les huit threads. Notre application a une sensibilité extrême à la latence et dépend plus essentiellement sur les threads. Il est donc important d'isoler ces threads et répartir les FFTs sur des noyaux séparés. D'autre part, nous allons mesurer le temps des huit FFTs indépendantes sur les huit threads.

Sous Linux, on peut désactiver les interruptions sur les noyaux. Il y a des fonctions du noyau qui nécessitent la présence des interruptions telles que l'interruption liée au compteur ordinal et le Read Copy Update (RCU) qui est responsable à la synchronisation. Ces fonctions peuvent produire une période d'attente entre les noyaux qui diminuera la performance ; cela peut conduire à produire des *Interruptions interprocesseurs* (IPI) entre les threads, même si ces interruptions sont petites et minuscules. La meilleure façon de résoudre ce problème est d'utiliser l'isolement des noyaux sur Linux pour réduire les changements de contexte. Alors que les threads utilisés par une application DPDK sont épinglés aux cœurs logiques sur le système, il est possible pour l'ordonnanceur d'exécuter d'autres tâches sur ces cœurs aussi. Pour éviter des charges de travail supplémentaires, il est possible d'utiliser le *isolcpus* sur les noyaux pour isoler les cœurs en ajoutant le paramètre *isolcpus* = [threads_number]. Cela informera l'ordonnancer du système d'exploitation de ne pas exécuter une tâche sur CPU, seulement quand l'utilisateur lui demande. La première partie de l'algorithme est de configurer l'environnement DPDK.

Gestion de la mémoire à l'intérieur du noyau avec DPDK

Dans un premier lieu, DPDK appelle la routine "rte_eal_hugepage_init()" pour initialiser les *Hugepages* dans des certains fichiers privés. Ensuite, il met à jour la routine

"rte_mem_config" et le fichier d'informations de *Hugepages*. DPDK fait le mapping afin de former des blocs adjacents dans l'espace mémoire virtuelle.

rte_config_init : Cette routine prépare la structure "rte_config" avec la configuration de la mémoire partagée. Elle commande des nombreuses structures telles que "rte_eal_memory_init ()", " rte_eal_logs_init ()" et " rte_eal_pci_init ()". Elle crée la configuration de la mémoire dans la mémoire partagée et dessine ce qu'il est écrit sur les segments de mémoire.

"eal_thread_init_master ()": Elle choisit le thread maître. Pendant ce temps "eal_thread_loop ()" commence à initialiser tous les autres threads. La routine "lcore_config" enregistre le "thread_id" dans une table avec les autres configurations du noyau.

"Pthread_create ()" : Elle démarre un nouveau thread dans le processus d'appel. En même temps, chaque thread va entrer dans la phase d'initialisation, alors nous appelons rte_eal_remote_launch () pour chaque thread et on le démarre (Figure 3-1). Les arguments reçus pour cette fonction sont la FFT de 2048 points, le numéro de thread "TID" et le choix si nous voulons exécuter la tâche sur le thread maître ou le thread esclave.

Après les initialisations du Kernel et DPDK, les threads seront créés. Une vérification est émise à partir du thread maître. Les threads esclaves attendent les uns des autres pour envoyer simultanément une réponse au maître. Enfin, tous les threads exécutent les FFTs.

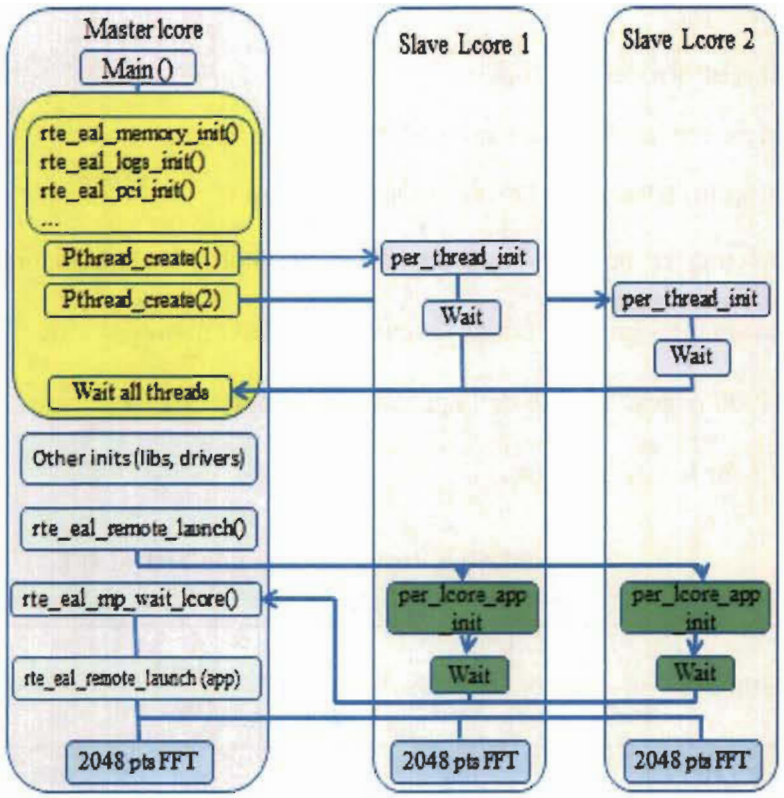


Figure 3.1 Organigramme de *multitasking* sur DPDK [10][11][12]

Dans la fenêtre de commande Linux, nous passons par différentes étapes pour permettre l'exécution de notre application :

1/ Ajouter les chemins d'accès des fichiers .h et les bibliothèques MKL aux variables d'environnement.

LIB=-L/media/5449-EF96/composer_xe_2013_sp1.2.144/compiler/lib/intel64/

INC=-I/media/5449-EF96/composer_xe_2013_sp1.2.144/mkl/include/

2/ Introduire les bibliothèques statiques MKL dans le Makefile + MKL FFT header

Include $(LIB)/libmkl_intel_ilp64.a$(LIB)/libmkl_core.a $(LIB)/libmkl_intel_thread.a $(INC)/mkl_dfti.h

3/ Ajouter l'environnement DPDK principal et les variables RTE_Target and RTE_SDK

export RTE_TARGET=x86_64-native-linuxapp-gcc

export RTE_SDK=/home/mounir/Desktop/dpdk17

4/ Activer les *Hugepages*

mkdir -p /mnt/huge

mount -t hugetlbfs nodev /mnt/huge

5/ Compiler et recevoir les résultats sur un thread (Master Thread)

gcc -Wall $(LIB) $(INC) -c FFT2048.c ./build/FFT -c 01 -n 1

Pour exécuter ce nombre élevé de FFTs dans notre étude statistique, les *Hugepages* offrent un avantage significatif dans l'organisation de la mémoire virtuelle. La mémoire est divisée en 4000 pages, la taille de la page est augmentée à 2 Mb. Nous avons fixé 2 Go de *Hugepages* pour le noyau Linux.

Tableau 6 Résultats pour une FFT (1×FFT)

Méthodes	Min (µs)	Max (µs)	Mean (µs)	σ^2	κ
FFT MKL	7	43	8	0.12	471
FFTW	6	65	7	1.12	938
FFT MKL DPDK	3	55	4	0.33	188.57
FFTW DPDK	3	67	6	0.31	291.52

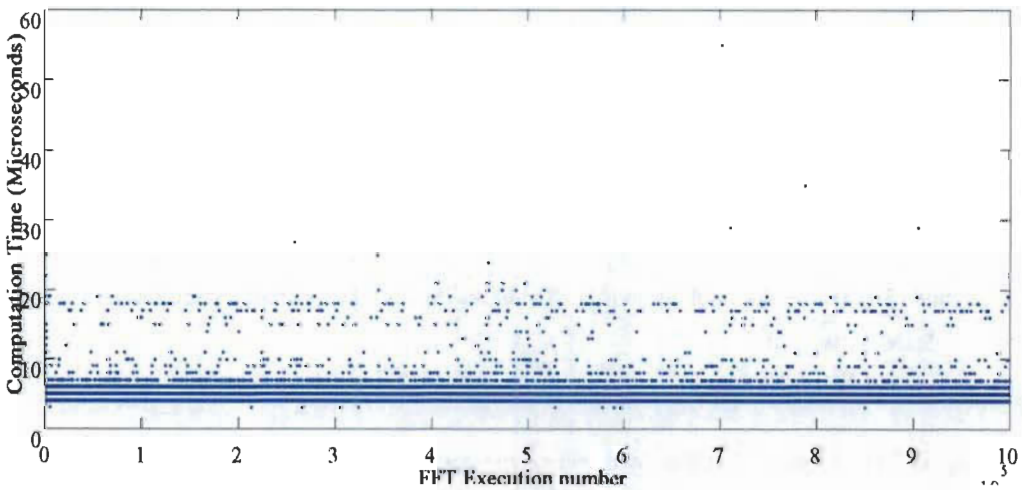


Figure 3.2 Une seule FFT MKL sur DPDK

Tableau 7 Résultats pour des FFTs indépendantes

Méthodes	Min (µs)	Max (µs)	Mean (µs)	σ^2	κ
4×FFTW	15	875	20	2.32	125510
4×FFTMKL	12	6932	47	193.66	68311
8×FFTW	16	20342	214	33251	3659
8×FFTMKL	15	623	29	5.08	21747

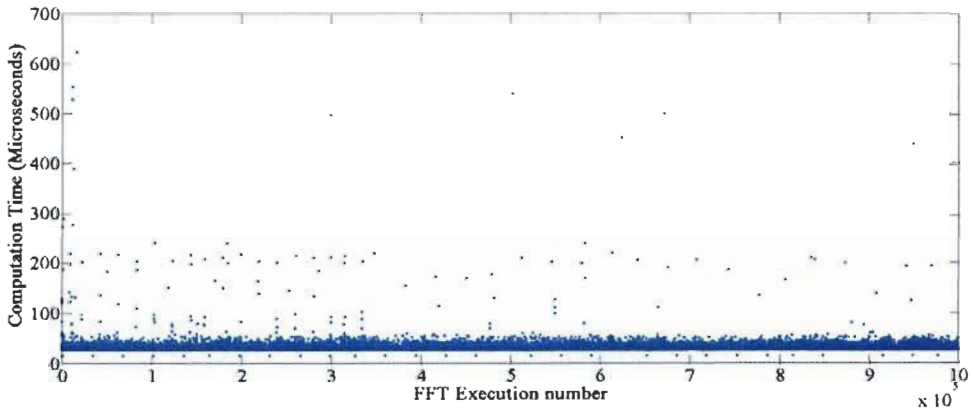


Figure 3-3 8 FFT MKL sur DPDK

3.3 Discussion

Les résultats d'implémentation (Tableau 6 et Figure 3-2) pour une seule FFT montrent que l'utilisation de DPDK présente de nombreux avantages à l'aide de l'isolation des cœurs. Cela nous a permis de réduire le temps de calcul d'un minimum de 7 et 6 µs à 3 µs. Pourtant, avec à peu près le même maximum. L'amélioration principale réside dans le temps moyen qui descend de 8 et 7 µs (MKL et FFTW) à 4 et 6 µs. En outre, le coefficient de kurtosis, qui présente l'uniformité des résultats, a connu une diminution de 471 et 938 à 188 et 291 (MKL et FFTW). La variance est mieux avec FFTW qui est ramenée de 1,12 à

0,31. Avec DPDK, nous avons réussi à affiner les résultats statistiques et de réduire le temps d'exécution moyen sur Intel Core i7 en isolant les noyaux sur le kernel Linux.

Les résultats d'implémentation pour plusieurs FFT (Tableau 7 et Figure 3-3), ont démontré une mauvaise performance, avec un grand max de 623 µs avec 8xFFT MKl et une haute valeur de kurtosis de 21747.

3.4 Conclusion

Dans cette partie du projet, nous avons implémenté l'algorithme de FFTs indépendantes avec et sans DPDK. Les résultats obtenus, principalement en terme de temps écoulé sont satisfaisants pour FFTW et MKL pour une seule FFT où nous avons réussi à réduire le temps moyen et la valeur de kurtosis en utilisant la technique d'isolement de noyau avec les *Hugepages*. Pourtant, les résultats pour des huit FFTs indépendantes n'étaient pas dans les normes.

Chapitre 4 - Les FFTs indépendantes avec OpenMP

4.1 Introduction

Ce travail étudie l'utilisation de processeurs Multicœurs/Manycœurs pour calculer des nombreux FFTs de différentes tailles comme indiqué dans la version 8 de la norme LTE pour toutes les largeurs de canaux spécifiés (1,25 MHz à 20 MHz), la taille de la FFT (128 à 2048 points) et les fréquences d'échantillonnage (1,92 MHz à 30,72 MHz) [5][3]. Comme le montre le tableau 2, la couche physique LTE utilise un certain nombre de blocs de ressources (RB) qui dépend de la bande passante du canal [3]. Ainsi, en considérant que pour chaque RB, une FFT est exécutée, dans le cas d'une largeur de bande de canal de 20 MHz, 100 RB sont utilisés, impliquant 100 FFTs de taille 2048 simultanément. Le calcul de FFT au sein du LTE sera parallélisé sur le Intel Core i7, Xeon et Xeon-Phi afin d'améliorer les performances de temps de calcul.

4.2 Des FFTs indépendantes et l'affinité des threads

Notre travail introduit la technique de parallélisation sur Parallel Studio XE 2013, avec l'utilisation de la bibliothèque OpenMP. Nous expliquons la différence entre les deux types de FFTs que nous avons testés : La FFT Multithreaded et les FFTs indépendantes. Pour le mode Multithreaded, nous avons effectué une seule FFT en demandant au compilateur d'utiliser tous les threads pour l'exécution. Pour les FFTs indépendants, nous avons effectué plusieurs FFTs sur plusieurs threads en contrôlant l'affinité avec OpenMP [37]. La bibliothèque OpenMP accélère le calcul des FFTs grâce à la KMP_AFFINITY [38], ce qui

aide à contrôler le nombre de threads par cœur à utiliser dans l'application. En fait, cela dépend de la topologie de la machine, qui est, s'il existe deux threads par cœur, ou quatre threads par cœur, la vitesse de traitement et la mémoire sont de type "distribué ou partagé". L'affinité des threads a une influence positive sur la performance de l'application. Il y a deux facteurs importants pour l'affinité avec OpenMP ; L'affinité dicte le nombre de threads à utiliser et reconnaît l'ID de tous les threads disponibles [38]. Le programme avec OpenMP commence son exécution dans un mode séquentiel en utilisant seulement un thread jusqu'à ce qu'il rencontre le pragma "*#pragma omp parallèle*". Dans cette région, le thread séquentiel devient le maître, et une zone mémoire est créée. Dans cette déclaration de pragma, tous les FFTs indépendantes seront exécutées en parallèle [37]. A la fin de l'exécution en parallèle, le premier arrivant des threads dans l'équipe attend tous les autres threads. Le groupe est dissout et nous retournons à l'exécution séquentielle (Figure 4-1). Dans notre application, seul le temps d'exécution de la région parallèle OpenMP est mesuré.

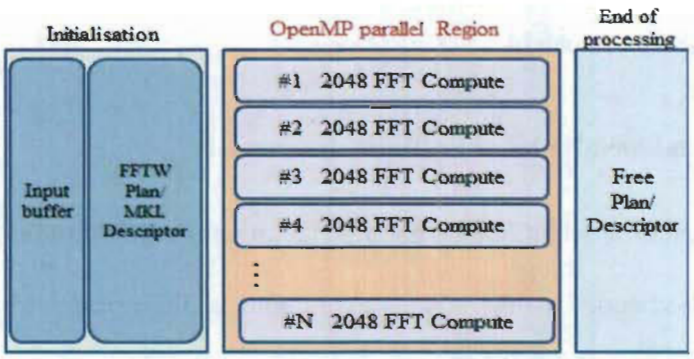


Figure 4.1 8 FFTs indépendantes sur Intel i7 (8 threads)

Avec les 8 threads sur Intel Core i7, nous commençons par récupérer le nombre de threads disponibles sur la plateforme, qui est 8 ; chaque thread a son propre numéro d'identification "ID", qui va de 0 à 7. Lorsque les données sont disponibles pour les threads

à exécuter, l'ordre des ID sera aléatoire. Quand nous commençons une région parallèle, la procédure est comparable à une course de chevaux où les chevaux sont les threads en cours d'exécution dans des directions aléatoires [37]. Les tâches de la FFT devraient être proportionnelles au numéro d'identification des threads.

Chaque thread démarre simultanément l'exécution de son secteur de longueur "*len*" dans le grand tableau "*array_temp*".

*FFT[ID]=&array_temp[ID * len]*

Si on prend len=2048 et len =8, les 8 FFTs indépendantes sont gérées ensemble comme suit.

*FFT0 = &array_temp[0*2048]*
...
*FFT7 = &array_temp[7*2048]*

Lorsqu'ID 0 termine, ses données ont déjà été écrites dans un tampon. Pendant ce processus, les IDs 1 à 7 continuent à travailler. Lorsqu'ID 1 termine sa tâche, les IDs 2 à 7 continuent à travailler. La même chose pour l'ID 2, qui termine sa tâche lorsque l'écriture d'ID 1 est terminé, et ainsi de suite [39]. La même méthode est utilisée avec le processeur Sandy-Bridge Xeon et le coprocesseur Xeon-Phi.

4.3 Les modes d'implémentation sur Xeon-Phi

Nous avons utilisé deux techniques d'affinité différente pour contrôler les threads. La première affinité est "*Compact*" qui maintient tous les traitements sur un seul processeur. Ceci est quand tous les threads du programme doivent entrer dans les différentes parties d'un grand tableau à plusieurs reprises. Tous les noyaux sur le même processeur pénètrent

dans les banques de mémoire de ce processeur [39]. Quand nos threads accèdent aux données enregistrées dans la mémoire d'un seul processeur, il est conseillé de les placer sur le processeur qui héberge cette zone mémoire. La seconde affinité est "*Scatter*" ou "*Round-Robin*". Cette affinité teste si les threads sont indépendants et n'ont pas un accès à plusieurs zones de mémoire contrairement au mode "*Compact*" [40]. L'avantage de cette affinité est que tous les threads doivent accéder et partager les mêmes secteurs de mémoire cache. Par conséquent, la latence de la mémoire est plus élevée lorsque les threads entrent dans une zone mémoire occupée par un autre processeur.

Pour Xeon-Phi, il existe deux méthodes évidentes pour gérer la mémoire non partagée entre un coprocesseur Xeon-Phi et le processeur Xeon. La première est l'organisation des données, où le compilateur crée et envoie les données du processeur au coprocesseur. La deuxième méthode est le modèle de mémoire partagée virtuelle, qui dépend de l'appui de l'exécution au niveau du système pour préserver la cohérence des données entre la mémoire partagée virtuelle de Xeon-Phi et Xeon [39]. La première méthode est fournie par OpenMP est plus fréquente que la seconde. C'est la méthode employée dans ce travail.

4.4 Évaluation de performance et résultats

Le 1 × SMFFT est une FFT Multithreaded séquentielle qui utilise les threads de toute la plateforme où le compilateur alloue des threads au hasard. La méthode proposée dans cette partie présente le P×PMFFT qui présente P FFTs indépendantes fonctionnant sur P threads. On a utilisé les noms des méthodes suivantes dans les résultats :

- 1×SMFFT – une FFT séquentielle Multithreaded
- *P*×PMFFT – *P* FFTs indépendantes, parallèles et Multithreaded

Le résultat idéal présente un temps de calcul prévisible. Les tableaux 8 à 10 montrent les indicateurs statistiques. Les valeurs d'asymétrie présentent un temps de calcul plus prévisible et une distribution gaussienne proche de la valeur moyenne [21].

Tableau 8 Temps de calcul des FFTs sur Intel i7-4770

Méthodes	Affinité	Min (µs)	Max (µs)	Mean (µs)	CV	α	κ
1×SMFFT MKL	Sans	7	43	7.9	0.044	8.9	471
8×PMFFT MKL	Scatter	14	100	15.3	0.097	26.8	1044
8×PMFFT MKL	Compact	14	150	15.2	0.104	32.4	1644

La performance sur Intel Core i7 est montrée dans le tableau 8. La FFT Multithreaded génère des nombreux temps d'exécution aberrantes et une haute valeur du kurtosis. Avec les FFTs indépendantes, nous observons que la FFT MKL a un maximum plus élevé avec l'affinité compact dû aux erreurs de partage.

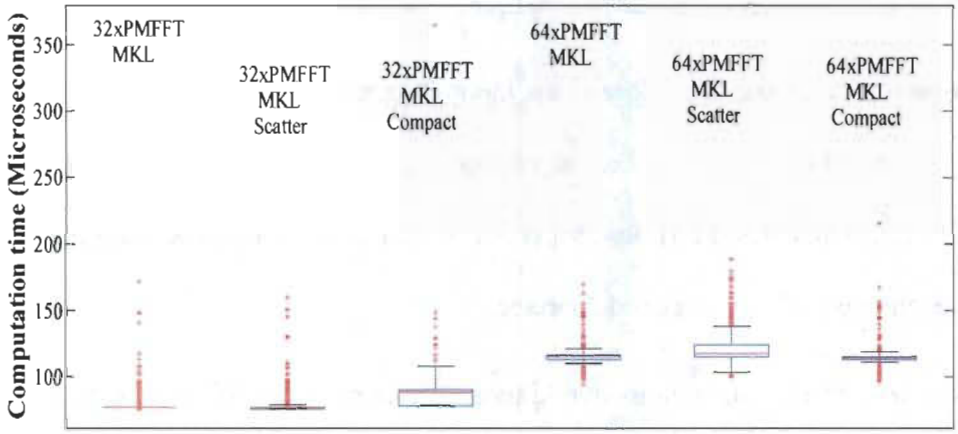
Pour le core i7, nous notons que le temps de calcul est 4 fois plus rapide en parallèle que séquentielle. Avec Xeon, le tableau 9 et la figure 4-2 présentent les valeurs statistiques et le boxplot, respectivement. Nous observons que MKL fonctionne bien pour l'implémentation des FFTs indépendantes. Pour Intel Core i7 et Xeon, on observe un gain en temps de calcul moyen lorsque le nombre de FFTs indépendantes augmente.

Le temps de calcul de 1×SMFFT avec les directives OpenMP est lent, même si le nombre maximal de threads est réglé sur 8 ou 16. En fait, les threads communiquent entre eux et accèdent à la même zone mémoire, parce que les entrées et les sorties sont définies comme partagées [16]. Voilà l'inconvénient de la mémoire partagée.

Tableau 9 Temps de calcul des FFTs sur Intel Xeon pour N=2048

Méthodes	Affinité	Min (µs)	Max (µs)	Mean (µs)	CV	α	κ
1×SMFFT MKL	Sans	16	96	26.9	0.238	5.3	36.3
16×PMFFT MKL	Scatter	16	77	18.7	0.291	6.8	52.9
16×PMFFT MKL	Compact	16	42	17.9	0.101	6.3	59.1

a)



b)

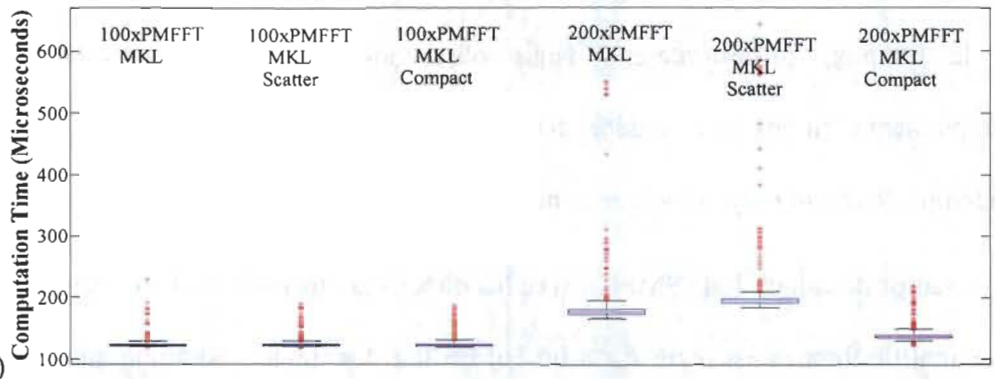


Figure 4.2 Boxplot des temps d'exécutions pour les FFTs parallèles (N=2048) sur Xeon-Phi pour a) 32, 64, b) 100 et 200 FFTs parallèles.

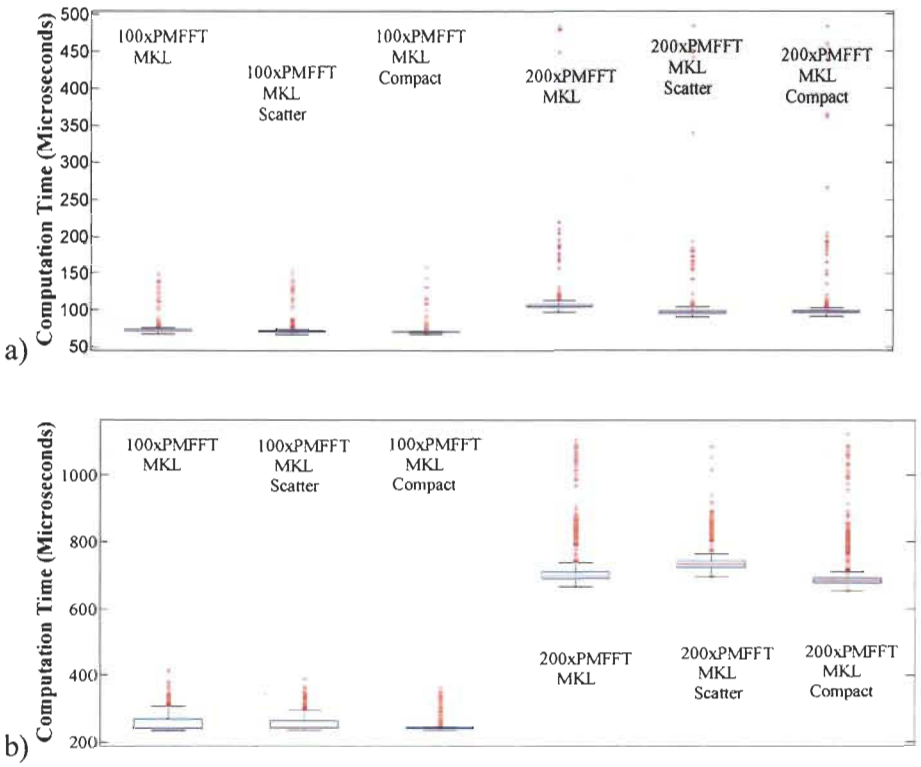


Figure 4.3 Boxplot des temps d'exécution pour les FFTs parallèles sur Xeon-Phi pour 100 et 200 FFTs parallèles avec a) N=1024 et b) N=4096

Le Xeon-Phi fournit des meilleurs résultats d'analyse statistique pour un grand nombre des FFTs indépendantes. Selon le tableau 10 et la figure 4.3a, lorsque le nombre des FFTs indépendantes augmente, l'affinité "*Compact*" fonctionne mieux que "*Scatter*". Le temps moyen de 200×SMFFT MKL sur Intel Xeon est de 5400 µs et le temps 200×PMFFT MKL sur Xeon-Phi avec l'affinité "*Compact*" est de 140 µs, effectivement nous avons une accélération du temps de 38 fois. Le gain en temps de calcul augmente avec le nombre des FFTs indépendantes et la taille de la FFT. Les bons résultats sur Xeon-Phi nous ont encouragés à doubler la taille d'entrée à 4096 (Figure 4.3b). Les résultats étaient uniformes avec des faibles valeurs d'asymétrie et de kurtosis.

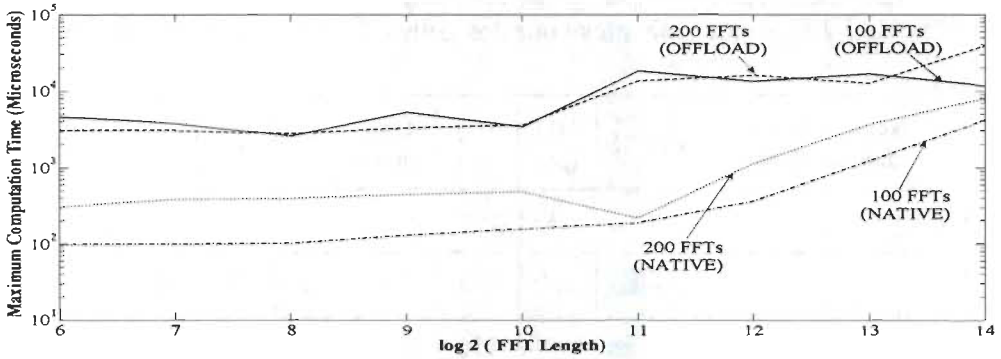
La figure 4-4 montre la différence entre les modes *Native* et *Offload* sur Xeon-Phi pour 100 et 200 FFTs en utilisant l'affinité "*Compact*". Pour observer la performance, nous explorons la taille de la FFT au-delà de la norme LTE. Ce chiffre révèle le temps moyen, le maximum et le coefficient de variation pour des différentes tailles de FFT allant de 64 à 16384. Tous les résultats sont à l'échelle logarithmique. D'une part, il est clair que le mode *Native* est préférable, car il a un maximum et un temps moyen de calcul inférieurs pour toutes les différentes tailles de FFT. La mesure de la dispersion semble être mieux en mode *Native*. D'une autre part, plus on augmente la taille de la FFT, plus le temps d'exécution s'améliore.

4.5 Conclusion

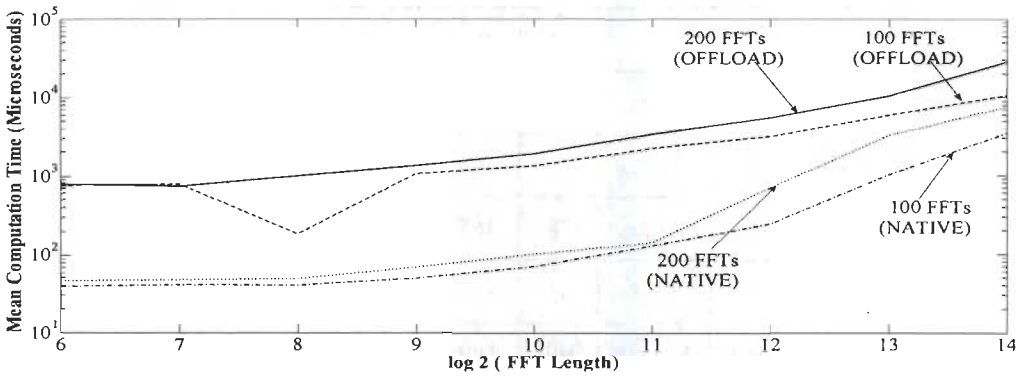
Dans ce travail, nous avons étudié l'utilisation des processeurs Multicœurs/Manycœurs pour calculer les FFTs indépendantes pour OFDM. Les valeurs statistiques montrent que l'exécution des grandes tailles de FFT implémentées sur les plateformes Manycœurs est beaucoup mieux que l'implémentation de petite FFTs sur le processeur Multicœur. Ce travail est une contribution à la mise en œuvre de la technologie LTE sur les plateformes Multicœur/Manycœur pour les réseaux C-RAN.

Tableau 10 Le temps d'exécution sur Intel Xeon-Phi pour P×PMFFT MKL pour l'exécution de P FFTs indépendantes pour des tailles de FFTs de 1024, 2048 et 4096 points

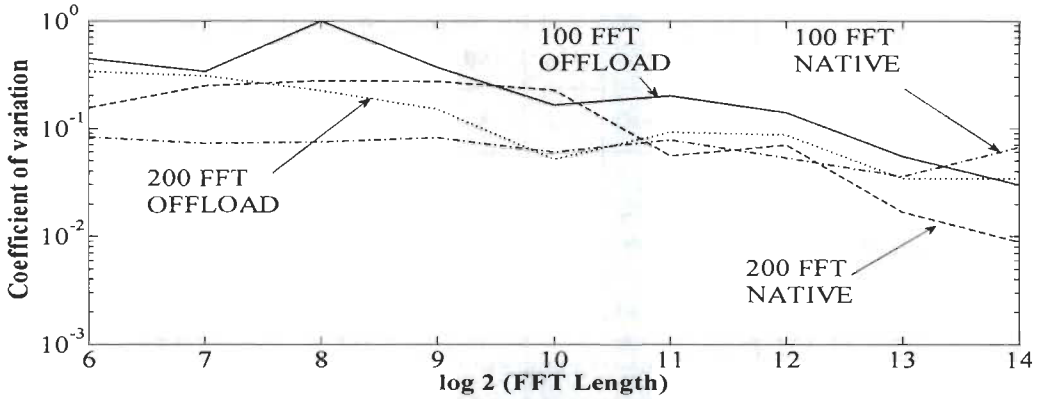
Nombre de FFTs indépendantes	Affinité	Min (µs)	Max (µs)	Mean (µs)	CV	α	κ
N=1024							
100×PMFFT MKL	Scatter	67	152	72.0	0.091	5.5	49.1
	Compact	67	156	69.9	0.060	12.6	198
200×PMFFT MKL	Scatter	91	483	98.5	0.185	16.5	310
	Compact	92	482	99.9	0.227	13.6	204
N=2048							
32×PMFFT MKL	Sans	76	172	78.9	0.063	8.7	116
	Scatter	76	160	78.9	0.069	7.4	80.9
	Compact	78	365	87.2	0.103	15.2	458
64×PMFFT MKL	Sans	95	170	115	0.048	2.0	21.8
	Scatter	100	189	122	0.100	1.3	4.7
	Compact	97	216	115	0.050	4.5	65.6
100×PMFFT MKL	Sans	119	230	124	0.048	8.5	98.2
	Scatter	118	190	124	0.043	7.9	78.1
	Compact	119	188	128	0.078	2.3	9.0
200×PMFFT MKL	Sans	166	552	182	0.151	10.7	133
	Scatter	184	644	200	0.159	10.2	117
	Compact	124	220	140	0.056	5.0	38.6
N=4096							
100×PMFFT MKL	Sans	234	409	255	0.105	1.6	4.5
	Scatter	236	385	256	0.099	1.6	4.4
	Compact	236	359	243	0.054	5.6	36.9
200×PMFFT MKL	Sans	667	1103	710	0.073	4.4	25.9
	Scatter	694	1084	740	0.045	3.4	19.5
	Compact	653	1121	694	0.071	4.8	31.0



a)



b)



c)

Figure 4.4 Comparaison entre les deux modes sur Xeon-Phi (Native and Offload) pour 100 et 200 FFTs indépendantes avec l'affinité Compact: temps de calcul maximal (a) et moyen (b) et CV (c).

Chapitre 5 - L'implémentation parallèle des FFTs indépendantes avec MPI

5.1 Introduction

Avec l'évolution des plateformes parallèles et les outils logiciels, les programmeurs confrontent le défi de choisir les architectures les plus adaptées pour leurs taches. De nombreuses techniques ont été proposées pour utiliser le parallélisme avec les types de mémoires ; partagées et distribuées. Dans cette partie, on a eu recours au MPI pour accélérer les performances des FFTs indépendantes afin de respecter la norme LTE.

5.2 L'implémentation parallèle et les outils logiciels

5.2.1 Les modes d'Implementation sur Xeon-Phi

Il existe deux modes importants d'implémentation sur Xeon-Phi : *Offload* et *Native*. Dans le mode *Offload*, le processeur exécute le programme et quand il rencontre une zone parallèle, il l'envoie au Xeon-Phi. L'inconvénient avec ce mode réside dans le long temps de téléchargement des données entre le processeur et le Xeon-Phi qui affecte la performance du programme [19]. Avec le mode *Native*, tout le programme s'exécute sur le Xeon-Phi. Le mode *Native* est le plus efficace et il est utilisé dans ce travail.

5.2.2 Le modèle Fork-joint avec Open MultiProcessing (OPENMP)

Dans ce mémoire, les FFTs indépendantes seront exécutées en parallèle avec une aucune dépendance des données entre eux. Pour paralléliser ce traitement, nous avons

besoin d'utiliser le modèle *Fork-Joint*, qui place un enchainement d'activités en parallèle. Pour bénéficier du ce modèle, OpenMP propose plusieurs fonctionnalités. Pour commencer une tâche, nous avons besoin d'utiliser la directive "*omp_pragma*". Lorsque ce pragma est inclus, le code devient une tâche concurrente, et prêt à être traité avec les threads [32]. OpenMP permet d'exécuter plusieurs threads en parallèle sur un ou plusieurs processeurs et sur le Xeon-Phi. Ce modèle obtient le nombre de threads spécifiés pour exécuter les FFTs (figure 5-1). Les fonctions OpenMP telles que "*omp_get_thread_num*" et "*omp_get_max_threads*" sont utilisés comme suit :

```
#pragma omp parallel num_threads(nThread)//Parallel Region
{int myID=omp_get_thread_num ();//Get the threads ID
status = DftiComputeForward(hand, &x[myID*len]);}//FFTs
```

Pour compiler le code, nous avons utilisé le drapeau "*-openmp*". Pour mettre en évidence le nombre de threads à utiliser dans notre application. Nous avons exporté la variable d'environnement suivante "*OMP_NUM_THREADS = 100*" pour préciser le nombre des FFTs indépendantes à exécuter.

Pour compiler et implémenter le code sur Xeon-Phi en mode *Native*, nous avons utilisé les drapeaux *"-openmp, -mmic, -mkl*" :

```
icc -mmic -mkl -openmp fftmklnative.c -o native.MIC
micnativeloadex native.MIC -e "OMP_NUM_THREADS=100"
```

5.2.3 La modélisation parallèle avec MPI

MPI est utile pour l'envoi des messages entre les threads dans un réseau, en utilisant des programmes hautement parallèles. En fait, les messages MPI peuvent se déplacer rapidement à travers un réseau TCP/IP [32]. Le fichier exécutable de notre application doit

être livré à ces nœuds en copiant ce fichier sur le Xeon/Xeon-Phi. Cela peut être fait manuellement ou en utilisant un système de fichiers distribué dans les processeurs Intel. Afin d'exécuter un programme MPI, il faut utiliser la commande "*mpirun*", tout en indiquant le nom de processeur et les Xeon-Phis où nous voulons exécuter les FFTs indépendantes. L'administrateur du réseau fournit une adresse, par exemple (aw 4r13 n09 mic0), qui cible le Xeon-Phi numéro 0 sur le processeur d'adresse aw-4r13-n09. Nous devons également mentionner le nombre de processus MPI pour chaque plateforme. Le message dans notre cas est une FFT exécutée sur un seul processus. Le communicateur principal, qui implique tous les processus, est le *MPI_COMM_WORLD*. MPI fournit un ensemble de fonctions qui sont très utiles pour développer le code de MPI. La fonction *MPI_Init_thread (& argc, & argv)* est réglée pour initialiser les paramètres *argc* et *argv* de *MPI_Init_thread ()*, qui sont les paramètres de ligne de commande du langage C. Pendant l'exécution, MPI supprime tous les paramètres de la commande que la bibliothèque peut gérer de *argv*, et réduit le nombre de *argc*. Par conséquent, nous devons prendre soin de ces paramètres après avoir appelé *MPI_Init ()* pour obtenir le nombre de processus totaux et l'ID de processus actuel invoqué comme suit:

MPI_Comm_rank (MPI_COMM_WORLD, &rank);/*get current process id */
MPI_Comm_size (MPI_COMM_WORLD, &size);/*get number of processes */

Contrairement aux OpenMP, le code MPI n'est pas synchrone. Ainsi, les clusters Xeon/Xeon-Phi doivent évidemment utiliser la synchronisation MPI. Les processus sont désynchronisés dès le début avec l'utilisation de *MPI_Init()* qui change le temps d'exécution de chaque processus. La deuxième cause de désynchronisation est les interruptions du système d'exploitation, où les processus partagent le temps de CPU avec

les commandes MPI. Par conséquent, pour mesurer le temps réel de l'exécution des FFTs parallèles, une barrière doit être placé comme suit, avant et après le bloc des FFTs :

```
MPI_Barrier(MPI_COMM_WORLD);
mytime = MPI_Wtime();
{status = DftiComputeForward(hand,x);//FFTs compute
//printf( "FFT from processor %d" of %d, rank, size);}
mytime = MPI_Wtime() - mytime;
MPI_Barrier(MPI_COMM_WORLD);
```

La première fonction *MPI_Barrier ()* peut conduire à une mauvaise synchronisation des différents processus MPI. En effet, il peut arriver que certains de ces processus apparaissent tôt lors de la prochaine barrière, tandis que d'autres apparaissent un peu plus tard. Dans ce cas, les premiers processus doivent attendre les processus en retard. Avec MPI, il faut garantir que tous les processus quittent la barrière en même temps.

Chaque processus appellera la même fonction FFT, avec les mêmes arguments, et donc chaque processus aura le même temps moyen d'exécution. L'une des directives de MPI la plus utile pour résoudre le problème de mesure de temps s'appelle la "*Réduction*". Chaque processus envoie son propre temps d'exécution de son FFT a un seul processus. Ce dernier processus calcule le temps moyen d'exécution des toutes les FFTs indépendantes. Nous utilisons *MPI_Reduce ()* pour obtenir la somme de tout le temps des processus, puis on fait division par le nombre total de processus pour obtenir le temps moyen. Le processus maître s'occupe de calculer ce temps. Cette réduction MPI peut être utiliser pour des opérations telles que la sommation, la multiplication, calcul de min et de max :

```
MPI_Reduce(&time, &Mean, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
if (rank == 0) {Mean /= size;//Master Thread collects the data and
```

printf("%lf\n",Mean);}//compute the Mean
MPI_Finalize();//End of MPI parallel region

Il n'est pas rentable d'obtenir directement le temps d'exécution de chaque processus sans passer par la réduction MPI, parce que tout le programme est en parallèle et le temps de calcul de chaque processus varie considérablement.

Après le chargement des bibliothèques OpenMPI et MKL dans le MIC, il faut exporter les variables d'environnement utiles. On aura besoin de la mémoire partagée (shm) car nous utilisons plusieurs nœuds de calcul. Le protocole TCP/IP permet la configuration du réseau avec la commande *I_MPI_FABRICS = shm : tcp*. Il est important d'exporter la variable *I_MPI_MIC = 1* pour permettre l'utilisation du coprocesseur Xeon-Phi. Pour compiler et exécuter le code de MPI sur Xeon et Xeon-Phi, nous avons besoin de suivre les instructions ci-dessous :

mpiicc -mmic -mkl -o fftmpi.MIC fftmpi.c // Compile
scp fftmpi.MIC mic0:~/. // Copy the executable to MIC
mpirun -n 16 -host aw-4r13-n09 ./fftmpi //run on the Host Xeon
mpirun -n 100 -host aw-4r13-n09-mic0 -env LD_LIBRARY_PATH
$MIC_LD_LIBRARY_PATH ./fftmpi.MIC //run on 1 MIC

Sur un Xeon-Phi, l'option "*mmic*" doit être insérée pour travailler en mode *Native*. Après la compilation, les fichiers exécutables et les données doivent être transférées au coprocesseur en utilisant la commande "*scp*". Au cours de cette étape, les adresses IP de CPU et Xeon-Phi sont allouées en exportant ces variables d'environnement. Il est possible d'exécuter des processus MPI sur une plateforme hétérogène CPU/MIC en mode *Native*, ce qu'introduit notre contribution majeure dans ce mémoire. Les différentes vitesses du CPU et MIC peuvent créer des problèmes d'équilibrage de charge par ce que la fréquence

d'horloge n'est pas la même dans les deux plateformes. On a utilisé un détecteur des fuites de mémoire tels que le logiciel valgrind et le débogueur gdb pour détecter les fautes de segmentations entre les threads. Avec un ensemble de Xeon-Phis, nous n'avons aucun problème d'équilibrage de charge, parce que tous les noyaux fonctionnent à la même vitesse dans un mode symétrique. Pour utiliser ce mode symétrique sur 2 × MICs, l'exécutable MPI doit être copié dans les deux MICs et être exécuté sur les deux simultanément. L'équilibrage consiste à exécuter 50 FFTs sur chaque MIC:

mpirun -n 50 -host aw-4r13-n09-mic0 -env LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH ./fftmpi.MIC :-n 50 -host aw-4r13-n09-mic1 -env LD_LIBRA RY_PATH $MIC_LD_LIBRARY_PATH ./fftmpi.MIC //Run 2 MICS

L'option *-host* représente le nœud pour commencer MPI. L'option *-env* indique les variables d'environnement déjà initialisées, et *./fftmpi.MIC* représente le chemin d'accès d'exécutable. Les bibliothèques MKL et MPI se trouvent sur le processeur, alors il faut les charger vers le coprocesseur avec *LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH.* Les virgules sont utilisées pour séparer les différents nœuds. Dans le mode *Native*, chaque MIC est un nœud autonome, ce qui est utile pour empêcher les communications entre les FFTs indépendantes. Comme le Xeon-Phi peut être considéré comme un nœud, chaque nœud peut être divisé en deux catégories : Xeon-Phi et la plateforme hétérogène Xeon/Xeon-Phi.

5.3 La hybridation OpenMP + MPI

En mode hybride, illustré à la figure 5-1, nous utilisons MPI pour aller à travers les nœuds de calcul et OpenMP pour exécuter des tâches dans les cœurs de chaque nœuds. L'avantage réside dans l'utilisation des deux types de mémoires partagée/distribuée simultanément. L'inconvénient est que la communication *peer to peer* entre le Xeon et le Xeon-Phi doit passer par les réseaux virtuels. Cette opération est couteuse et peut ralentir la

communication physique entre les nœuds. Lors de l'écriture d'un programme avec MPI et OpenMP, certaines bases doivent être envisagées, y compris l'emplacement de stockage des variables, la méthode d'accès pour les processus et les threads et la façon dont ils échellent dans les noyaux. Nous introduisons OpenMP dans MPI avec la directive MPI_THREAD_FUNNELED pour assurer que le processus est Multithreaded, de sorte qu'il peut y avoir plusieurs OpenMP threads pour chaque processus MPI.

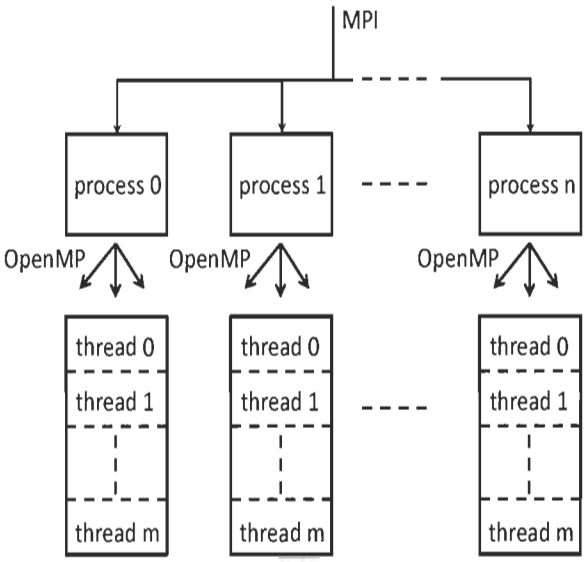


Figure 5.1 Le mode hybride sur n processus MPI et m OpenMP threads

La directive suivante garantit que seulement le processus maître dans MPI fait l'appel à la directive OpenMP :

```
int required=MPI_THREAD_FUNNELED;
int provided; // Provided level of MPI threading support
MPI_Init_thread(&argc, &argv, required, &provided);
```

Au lieu de *MPI_Init ()*, l'initialisation sera *MPI_Init_thread ()*. Nous appelons un pragma OpenMP qui construit une structure dans la région parallèle du MPI. Chaque FFT sera exécutée par un seul processus MPI et un nombre de threads OpenMP simultanément :

```
#pragma omp parallel
 { ... //FFT execution on OpenMP Threads
printf( "FFT from processor %d" of %d, thread %d\n", rank, size,
omp_get_thread_num()); } MPI_Finalize();
```

5.4 Evaluation du performance et résultats

5.4.1 Simulation

Dans chaque expérience, 2×10^3 FFTs de taille N ont été exécutés sur les Xeon-Phis. Pour analyser la distribution de 2×10^3 temps de calcul, nous avons considéré les paramètres statistiques suivantes ; la moyenne, max et coefficient de variation.

5.4.2 Discussion des résultats

Dans les graphiques de la figure 5-2, On explique les temps de calcul observés pour les sept méthodes d'exécution utilisées dans ce travail. Le plus grand temps a été observé avec l'hybridation OpenMP + MPI sur 1×MIC avec un temps moyen de 700 µs et un max de 800 µs. En utilisant l'implémentation hétérogène pour le mode hybride, on a réduit le temps moyen à 31,3 µs. Les résultats avec MPI pure sur les MICs ont montré que l'augmentation du nombre de MIC génère un meilleur temps d'exécution. Cependant, avec 3×MICs la valeur max de 93 µs est supérieur à notre objectif de 66,7 µs de LTE, même si son temps moyen de 54 µs est dans la zone de sécurité. La méthode avec OpenMP sur 1×MIC semble être dans une bonne plage de temps, mais elle ne satisfait pas aux exigences. Le meilleur temps d'exécution est obtenu avec MPI pure sur la plateforme hétérogène 3×MICs + CPU en profitant d'un bon équilibrage de la charge sur les noyaux. La figure 5-3 montre les valeurs moyennes, max, coefficient de variation (CV) des sept

différentes méthodes d'implémentation. L'implémentation avec MPI pure sur 3×MICs montre une bonne évolutivité pour les mémoires distribués dans les MICs. La bonne performance est obtenue grâce à l'optimisation et l'équilibrage de charge entre les cœurs et la synchronisation entre les nœuds. Lors de combinaison de la mémoire partagée/distribuée, la MPI pure sur le système hétérogène semble être le plus efficace que les autres méthodes. L'implémentation avec OpenMP a montré également une bonne performance, mais elle a un inconvénient majeur par rapport à MPI. En fait, la parallélisation avec OpenMP a besoin d'un espace d'adressage de mémoire partagée qui limite l'évolutivité sur le MIC. OpenMP fonctionne mieux sur une plateforme SMP. Sur la plateforme hétérogène MIC + CPU, l'interaction entre MPI et OpenMP est plus forte que dans MIC. Les appels de MPI sont effectués à partir de régions parallèles et les threads où les différents processus MPI doivent se synchroniser les uns avec les autres. Avec l'hybride OpenMP + MPI sur MIC, les threads OpenMP ont manqué l'efficacité de la mémoire partagée disponibles sur les plateformes SMP ce qu'a conduit à un temps d'exécution moyenne, max, et un coefficient de variation élevés.

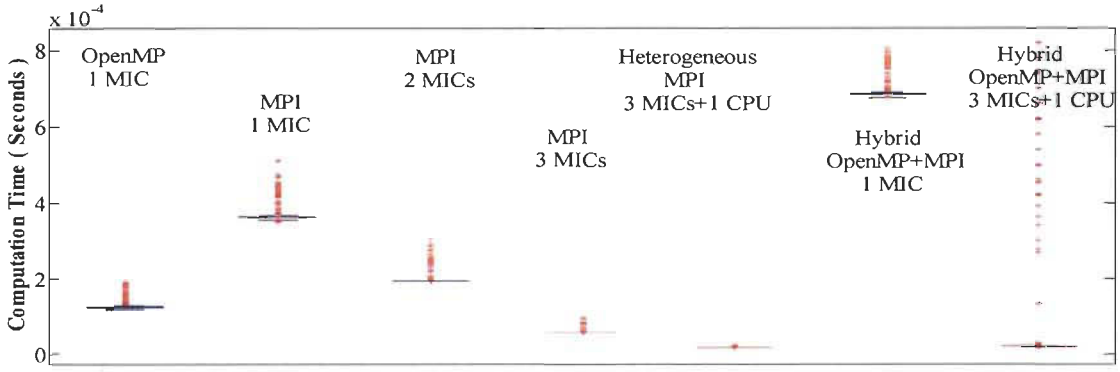


Figure 5.2 Boxplot du temps d'exécution pour100 FFTs concurentes (N=2048).

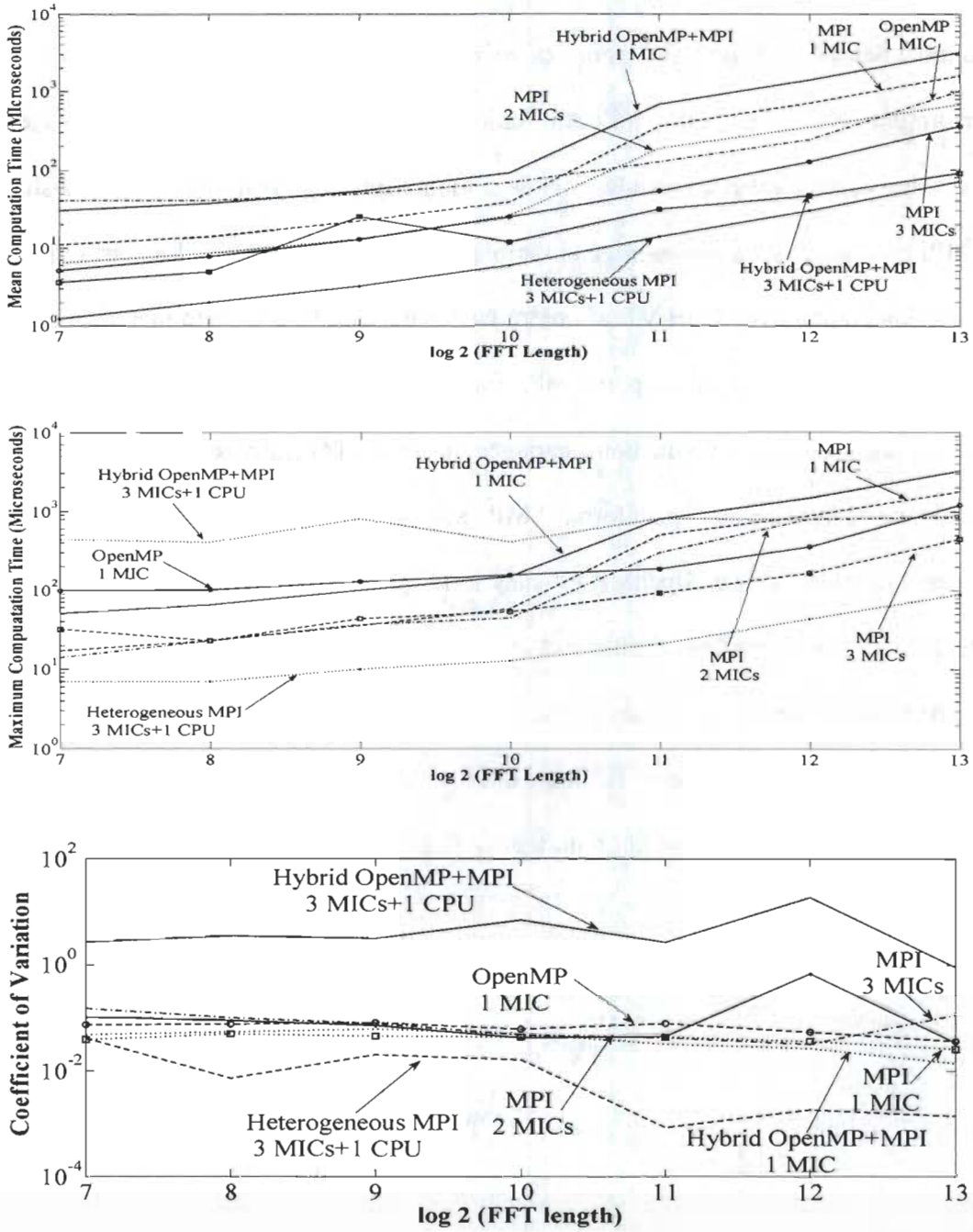


Figure 5.3 Comparaison des temps d'exécution dans les septs modes d'implémentation Xeon/Xeon-Phi pour 100 FFTs concurrentes: a) temps d'exécution moyenne b) temps max, et c) coefficient de variation.

Les valeurs de CV présentent des faibles valeurs inférieures à 0,01 pour toutes les tailles de FFTs avec MPI sur l'ensemble 3×MICs +CPU, contrairement à l'hybride OpenMP+MPI qui a dépassé 0,5. Les valeurs de CV avec MPI sur 1, 2 et 3 MICs et OpenMP restent proches de 0,1. L'hybride OpenMP+MPI crée une décomposition de domaine comme une application à deux niveaux. En fait, cette hybride OpenMP+MPI présente des défis difficiles à cause du placement des threads, alors que la MPI pure adapte facilement avec la topologie de notre plateforme.

5.5 Conclusion

Dans ce travail, nous avons exploré des divers moyens d'implémentation des 100 FFTs en utilisant OpenMP sur 1xMIC, MPI sur un, deux et trois MICs, MPI sur la plateforme hétérogène 3×MICs+CPU, l'hybride OpenMP + MPI sur 1×MIC et 3×MICs+CPU. La solution basée sur MPI pure exécutée sur une plateforme hétérogène de Xeon/Xeon-Phi a respecté la contrainte de LTE de 66,7 us et a bien utilisée l'architecture du système.

Chapitre 6 - La parallélisation d'une FFT

6.1 Introduction

La transformée de Fourier discrète (DFT) implique une large gamme d'applications. Une autre façon de calculer la transformée de Fourier discrète consiste à utiliser la transformée de Fourier rapide (FFT). La FFT unidimensionnelle (1-D) reçoit en entrée un ensemble de nombres complexes N et génère en sortie un tableau de taille N. Elle est utilisée dans le traitement numérique du signal pour passer du domaine temporel au domaine fréquentiel. Cependant, dans certaines applications le processus de conversion est très couteux en temps de calcul.

Ce travail couvre le concept traditionnel pour calculer la FFT en divisant l'ensemble du problème en sous-blocs parallèles et allouer les noyaux d'une façon parallèle pour obtenir une meilleure synchronisation et une meilleure performance. Elle rend compte de l'utilisation du Xeon-Phi (MIC) et CPU Xeon pour calculer une FFT très longue. Ce chapitre est une exploration de la mise en parallèle d'une seule FFT et va en dehors des normes LTE en ce qui concerne les longueurs de FFT ($2^7 < N < 2^{30}$). Ce travail étudie en temps réel la FFT parallèle en utilisant trois étapes (diviser pour régner, traitement et recombinaison). Une comparaison entre Intel MIC et GPGPU est réalisée afin d'extraire les différences entre les deux plateformes.

6.2 Radix-2 Cooley Tukey

L'algorithme de Cooley Tukey est introduit dans les architectures de la plupart des modèles de FFT [46]. Il est largement utilisé pour améliorer ses performances. Dans ce travail, nous employons seulement le calcul radix-2 pour simplifier le processus de parallélisation. Le concept de ce modèle à base 2 est de programmer l'équation (1) en tant que somme des composantes paires et impaires. Avec l'équation (2), on peut remarquer que la première somme est la FFT de nombres pairs et la seconde somme est la FFT de nombres impairs multiplié par un élément exponentiel. Cet élément exponentiel est appelé "Twiddle Factor" ou coefficients de Fourier, *W*.

$$\begin{aligned} F_k &= \sum_{n=0}^{N-1} x_n \,.\, \exp\left(-i\frac{2\pi kn}{N}\right) \\ &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \,.\, \exp\left(-i\frac{2\pi k(2n)}{N}\right) + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \,.\, \exp\left(-i\frac{2\pi k(2n+1)}{N}\right) \\ &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} \,.\, \exp\left(-i\frac{2\pi kn}{N/2}\right) + \exp\left(-i\frac{2\pi k}{N}\right)\sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} \,.\, \exp\left(-i\frac{2\pi kn}{N/2}\right) \end{aligned} \quad (1)$$

$$F_k = E_k + \exp\left(-i\frac{2\pi k}{N}\right) O_k = E_k + W.O_k \quad (2)$$

où

F_k: k-ième coefficient de la transformation

x_n: n-ième entrée des données

N : Longueur total de la FFT

Appelons la partie paire de la transformation "E_k" (*Even*) et la partie impaire "O_k" (*Odd*). Ces valeurs doivent être trouvées pour $0 < k < N/2$. Le reste des valeurs de k compris entre $N/2$ et N peuvent être extraits grâce à la périodicité et la symétrie des fonctions

exponentielles complexes. Le but de la phase de recombinaison est d'appliquer les équations (3) et (4) pour assembler les résultats de petites FFTs afin de produire le résultat de la plus grande FFT décomposée. Les deux équations (3) et (4) présentent les résultats de la FFT respectivement pour $0 < k < N/2$ et $N/2 < k < N$.

$$F_k = E_k + \exp\left(-i\frac{2\pi k}{N}\right) O_k \quad \text{pour } 0 < k < N/2 \tag{3}$$

$$F_{k+\frac{N}{2}} = E_k - \exp\left(-i\frac{2\pi k}{N}\right) O_k \quad \text{pour } N/2 \leq k < N \tag{4}$$

Le modèle Cooley Tukey reproduit cette décomposition récursive pour générer des sous-DFT. Ces sous-DFTs suivent le même concept de symétrie pour rendre le calcul de la FFT parallèle plus facile. Notre travail utilise la bibliothèque OpenMP parallèle avec l'implémentation des sous-FFTs MKL. Ces sous-FFTs MKL sont assez rapides pour obtenir des bonnes performances [1]. La longueur de la FFT de taille N peut être décomposée et redimensionnée sous la forme: $N = M \times B$ avec $B = 2^s$ avec M et B représentant la longueur de bin et le nombre des bins, respectivement, alors que s représente le nombre de fois que nous voulons faire une division récursive de type radix-2 de la FFT, appelé "split". Par exemple, avec s=2 splits, nous appliquons l'algorithme de la FFT 2 fois. Dans ce cas, nous divisons la transformée de taille total N par $B=2^2=4$, c'est à dire à M petites transformées de taille N/B (Figure 6-1).

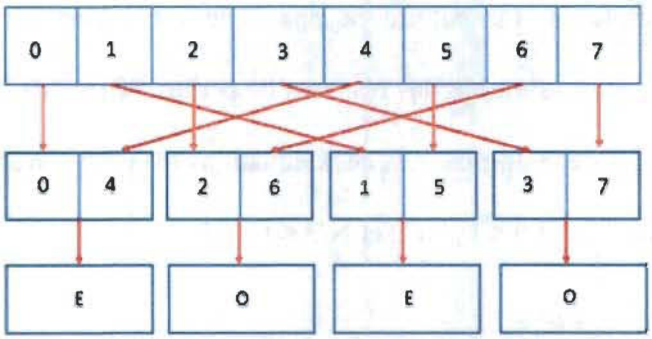


Figure 6.1 Hiérarchie de la FFT parallèle de taille 8 et s=2 splits.

6.3 Les étages de parallélisation

La première étape est la division de l'entrée N à de nombreux blocs suivis par un traitement des FFTs indépendantes en mono-thread. Nous finissons avec une phase de recombinaison.

6.3.1 L'étage de division

Processus d'inversion de bits

Au cours de cette étape, l'ordre des données d'entrées doit être converti à l'ordre binaire inverse (*bit inversing*). La modification des positions de données manuellement peut causer des fuites de mémoire. Ceci doit être évité au cours des opérations 1-D, lorsque les threads accèdent aux données de la mémoire partagée. Ensuite, chaque thread crée l'adresse de données de bit inversé et enregistre l'adresse dans sa mémoire [3]. Si l'on considère le cas où $N = 8$, nous savons que la première décimation donne la séquence suivante : x(0), x(2), x(4), x(6), x(1), x(3), x(5), x(7) et le second résultat de décimalisation de la séquence x(0), x(4), x(2), x(6), x(1), x(5), x(3), x(7). Cette commande d'entrée affiche la décimation de la séquence de huit points (Figure 6-2). Nous ne pouvons pas tout simplement décomposer les

paires et les impaires d'un grand tableau à l'aide d'une simple boucle. Pourtant, en parallèle la méthode des messages MPI restent le meilleur choix pour éviter le retard des transferts de données et la lente communication entre la mémoire cache et partagée.

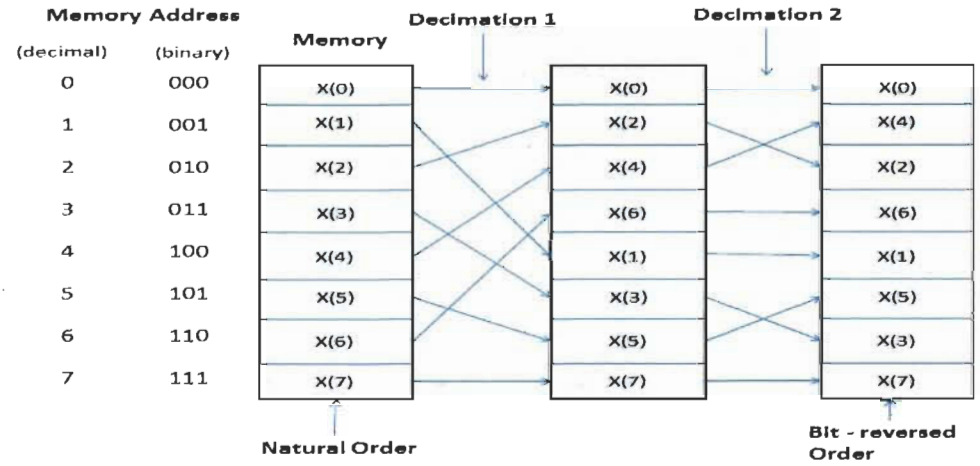


Figure 6.2 Processus d'inversion de bits [25]

Concept de diviser pour régner avec MPI et mapping

Chaque nœud utilise des blocs tampons afin qu'il puisse successivement envoyer et recevoir des messages plus petits dans ces blocs. Dans cette étape de diviser pour régner, nous avons besoin d'assurer l'efficacité parallèle et de partager les blocs à travers des processeurs. Les FFTs et les routines de remapping nous ont permis de préciser comment les blocs sont mappés aux processeurs sur l'entrée et la sortie [29]. La seule qualification pour un tel mapping est que chaque processeur possède une sous-partie du tableau 1-D qui est le bin (Figure 6-3). Sur chaque processeur, les blocs doivent être enregistrés d'une manière adjacente à la mémoire. Quand on regarde à partir de la sortie jusqu'à la fin du processus, toutes les données doivent être contiguës dans la mémoire. Fondamentalement, ces routines sont similaires à la façon MPI_Alltoall: chaque processeur dans un groupe

envoie un message à tous les autres processeurs. Ce groupe est un ensemble de processeurs en fonction de la hiérarchie des données [29].

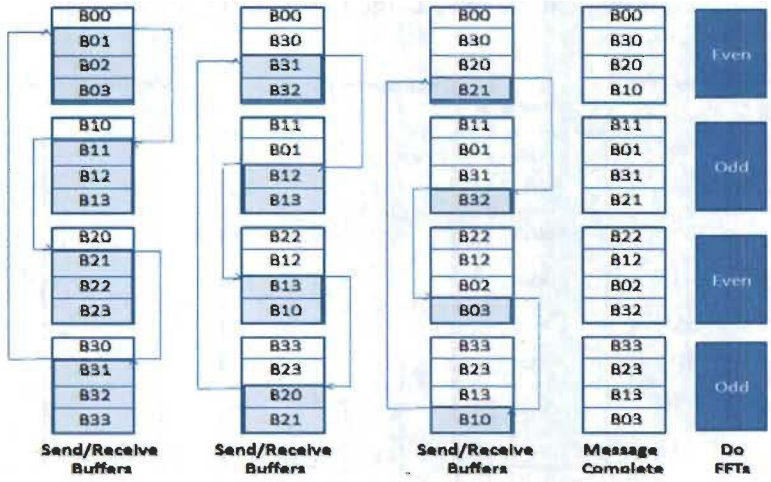


Figure 6.3 Dispersion avec MPI_Alltoall [29]

6.3.2 L'étage de traitement

Durant cette phase, nous exécutons les FFTs indépendantes et nous allons les mettre dans les blocs (Evens & Odds). Nous appliquons des FFTs indépendantes dans chaque bloc sur CPU/Xeon-Phi. Les FFTs indépendantes peuvent être exécutées en parallèle sans dépendances de données entre eux comme on a vu dans les chapitres précédents. Pour paralléliser ce traitement, nous avons besoin d'utiliser le modèle de calcul parallèle de Fork-Joint déjà étudiée.

6.3.3 L'étage de recombinaison

Cette phase utilise l'algorithme Radix-2 Cooley Tukey pour recombiner les résultats des petites FFTs générées dans l'étape de diviser pour régner. Nous appelons " E_k " et " O_k " les résultats de la FFT des composants pairs et impairs.

$E_k = FFT_k(Evens)$

$O_k = FFT_k(Odds)$

F_0 et $F_{N/2}$ sont des nombres réels qui doivent être calculés séparément. Les éléments de la FFT finale ont la forme de nombres complexes écrits sous la forme de l'équation (5), où R_k est la partie réelle et I_k est la partie imaginaire. W est le twiddle factor.

$$F_k = R_k + I_k \tag{5}$$

$$F_k = FFT_k(evens) + W.FFT_k(odds) \tag{6}$$

$$F_{k+N/2} = FFT_k(evens) - W.FFT_k(odds) \tag{7}$$

Les valeurs des FFT_K (Evens) et FFT_K (Odds) dans (6) et (7) sont calculées pour $0<k<N/2$. Grâce à la périodicité des fonctions exponentielles et trigonométriques qui permettent la symétrie, on peut calculer la FFT_K (Evens) et FFT_K (Odds) sur $N/2 \leq k<N$. Le traitement parallèle se fait en répartissant les FFTs indépendantes dans chaque cœurs de processeur.

6.4 Évaluation des résultats

6.4.1 Performance

L'équation(8) est utilisée pour calculer la performance d'exécution des FFTs de type complexes selon les travaux de Frigo&Johnson [49] exprimé en (GFLOP/s). Cette expression dépend essentiellement du nombre de points (N) et le temps d'exécution de la FFT en microsecondes [45].

$$\text{Performance(GFLOP/s)} = \frac{5 \times N \times \log2(N)}{\text{RunningTime(Microseconds)}} \tag{8}$$

Considérant le grand nombre de points de FFT ($N>2^{18}$), la mesure du temps d'exécution est pour une seule FFT. Il y a donc une seule réalisation de FFT pour chaque mesure du temps. Ce temps de calcul comprend seulement le traitement et la recombinaison, l'initialisation et l'étape de division est exclue du temps d'exécution.

6.4.2 Conditions de simulations et les plateformes utilisées

Les simulations ont été réalisées sur le Xeon et le Xeon-Phi. Le nombre d'itération par FFT dépend de la plateforme utilisée de telle sorte que chaque thread s'occupe d'un bin. L'initialisation de la FFTW, FFT MKL et la FFT parallèle ne sont pas inclus dans le temps de calcul. Le temps d'optimisation par *bit-reversing* n'est pas inclus dans les mesures. Cependant, il joue un rôle important dans le traitement des FFTs indépendantes et leurs placements en mémoire.

Sur Xeon, une FFTW ou FFT MKL *One-threaded* est une FFT qui utilise seulement un thread pour son exécution. Par contre, une FFT *Multithreaded* utilise la totalité des threads disponibles sur la plateforme (16 threads) (Figure 6.4.a).

La performance de la FFT parallèle varie avec le nombre de splits sur les deux plateformes Xeon et Xeon-Phi (Figure 6.4.b et Figure 6.4.c).

6.4.3 Discussion des résultats

Selon les mesures (Figure 6-4), la FFT parallèle a dépassé la FFTW pour N supérieur à 2^{18} et MKL pour N supérieur à 2^{19} avec une performance de 44 Gflop/s. Ce qui est 4 fois plus rapide quand N est égal à 2^{29}. Un Multithreaded MKL et FFTW performent bien avec les petits FFT $2^7<N<2^{18}$, mais ils perdent en performance en stagnant à 2 Gflop/s avec les grandes FFTs.

Le nombre de divisions ou les splits a un effet significatif sur la performance, car ce nombre est étroitement lié au nombre de cœurs et de threads de la plateforme. Si on prend splits = 5 sur le Xeon, on aura un nombre de bins, $B = 2^5 = 32$. À l'intérieur de chaque bin, nous avons ($M=N/B$). Chaque thread exécute 1 bin simultanément jusqu'à 16 (le nombre total des threads sur Xeon). Ensuite, ces threads exécutent les autres 16 bins. Nous ne devrions pas augmenter le nombre de splits arbitrairement. Si on prend splits = 6, nous aurons un nombre de bins, $B = 2^6 = 64 = 16\times4$. Donc, les threads vont calculer 4 fois séquentiellement pour terminer tous les 64 bins.

Il faut que le nombre de splits sur Xeon-Phi (Figure 6-4-c) ne soit pas le même que le Xeon parce que chaque plateforme a sa propre architecture et son nombre de cœurs. Pour les grandes FFTs, nous travaillons avec le nombre maximal de threads disponibles sur le système.

Nous avons 244 threads sur Xeon-Phi, de sorte que le nombre de bins, B, va être plus élevé que Xeon. Pour splits = 11, le nombre de bins, $B = 2^{11} = 2048$. Donc les threads feront 2048/244 = 8 tours pour terminer le calcul. Si on prend splits = 5 comme Xeon, alors on aura un nombre de bin B = 32. Donc la plateforme ne travaille pas à haute vitesse, 32/244 = 0,12 tour. Nous pouvons remarquer que le Xeon-Phi ne fonctionne pas avec son plein potentiel (seulement 12% des threads sont en service) ce que causera une perte de performance.

La performance de la FFT parallèle est plus petite que MKL et FFTW (Figure 6-4-a) pour un faible nombre d'entrée ($< 2^{19}$). Les résultats montrent que FFTW a réalisé une bonne performance, où N est compris entre 2^8 et 2^{20}. Nous pouvons remarquer une baisse de performance lorsque $N > 2^{18}$. Le problème de la FFTW est la présence de tenseurs via les

entrées/sorties (E/S) [45]. Ces tenseurs sont situés au niveau de leurs plans pour améliorer les performances pour les petits nombres d'entrées. Ils ont tendance à avoir des résultats optimaux pour de petites ($N<2^9$) et moyennes ($2^9<N<2^{18}$) longueurs de FFT. Si les tables des E/S ne sont pas adjacents dans la mémoire, des tampons sont utilisés pour augmenter les performances grâce à leur interaction avec la mémoire cache et le reste de mémoire dans le système [45]. Les boucles imbriquées et la méthode de vectorisation dans le FFTW et MKL restent inconnues.

Notre étude est basée sur la comparaison entre le Xeon-Phi et le GPGPU [25]. Les deux plateformes Xeon-Phi et GPGPU ne fonctionnent pas bien avec des faibles entrées (moins de 2^{20}). Prenons note de la notation utilisée, à savoir, la notation FFTW1 et FFTW16 signifie l'utilisation de FFTW respectivement sur un seul thread (One Threaded) et 16 threads (Multithreaded) sur le CPU Xeon.

Selon les résultats de la figure 6-4-a, nous avons une performance de 20 GFLOPS sur FFTW16 et seulement 7 GFLOPS pour la FFT parallèle pour $N=2^{17}$ points. Les résultats s'améliorent avec un nombre des entrées plus élevés (plus de 2^{20}).

Avec $N = 2^{26}$ ($64M$), ils ont trouvé un temps écoulé de leur FFT parallèle de 2,19 secondes sur GPGPU, 44,3 secondes de leur FFTW1 et 7,9 secondes de leur FFTW16 dans les figures 6 et 7 de leur article [25]. Cela permettra de remarquer que le GPGPU a réalisé un speed-up de 20 par rapport leur FFTW1 et 3,6 par rapport leur FFTW16. Ils n'ont pas fait l'implémentation de la FFT parallèle sur Xeon, car le CUDA C est utilisable seulement sur les GPUs [25].

Avec $N = 2^{26}$(64M), nous avons trouvé que notre FFT parallèle sur Xeon est 4 fois plus rapide que FFTW16 et 26 fois plus rapide que FFTW1. Sur Xeon-Phi, notre FFT parallèle

est 20 fois plus rapide que FFTW1 et 3 fois plus rapide que FFTW16. On peut remarquer la similitude entre les plateformes Xeon-Phi et GPGPU avec des gains très proches. Notre FFT parallèle sur Xeon a dépassé le gain de GPGPU en comparaison avec la FFTW1 et FFTW16.

6.5 Conclusion

Nous avons procédé à l'implémentation de la FFT parallèle sur Xeon et Xeon-Phi. D'après notre mesure, la FFT parallèle a dépassé la FFTW pour $N > 2^{18}$ et MKL pour $N > 2^{19}$ pour avoir une performance de 44 GFlop/s, ce qui est 4 fois plus vite que la FFTW16 pour un nombre de point $N = 2^{29}$. Les MKL et FFTW ont commencé bien avec des petites entrées, mais ils ont diminué à 2 GFlop/s avec des grandes FFTs. Les résultats de la comparaison ont prouvé que Xeon-Phi et GPGPU travaillent dans le même domaine de performance.

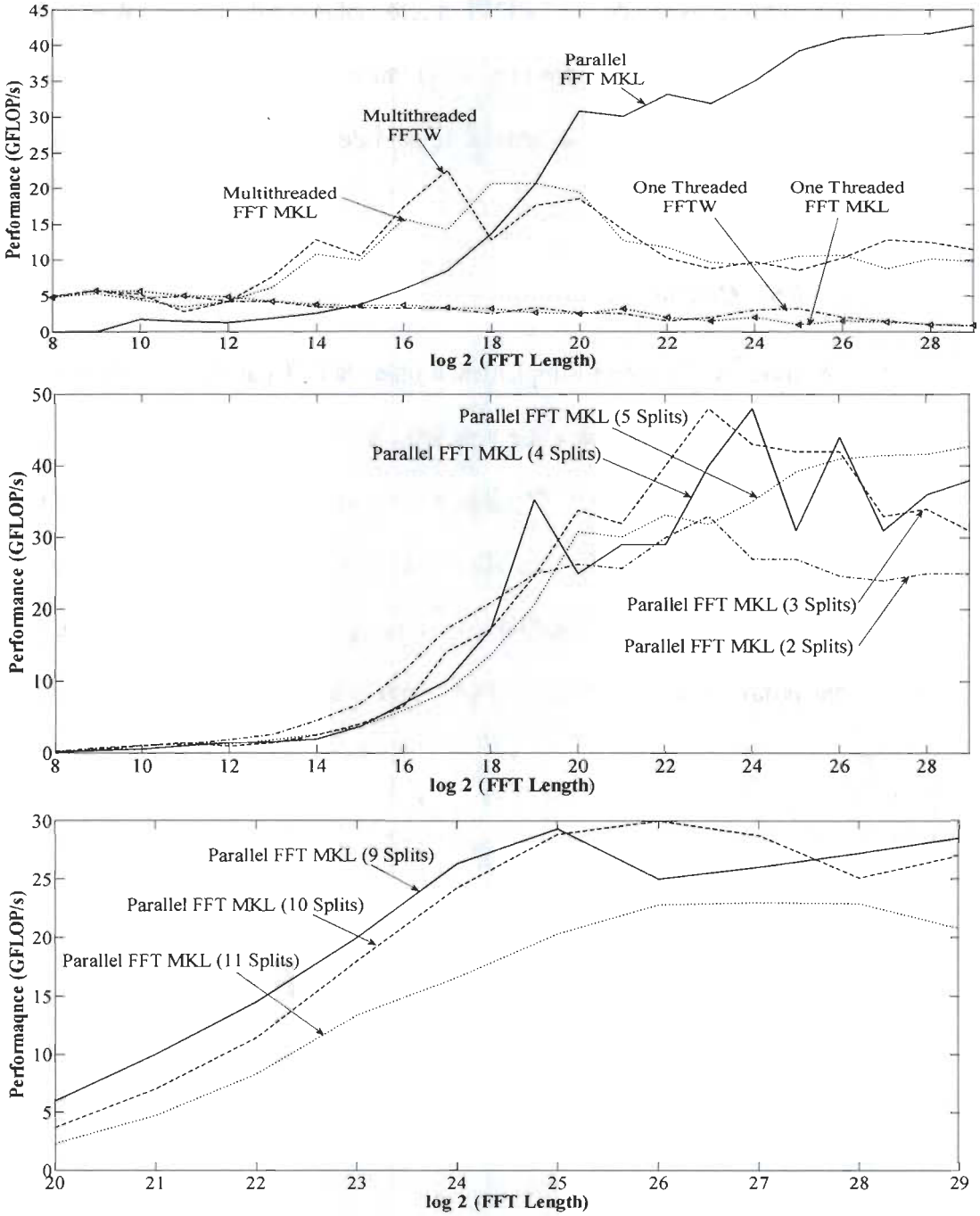


Figure 6.4 a) Performance sur CPU b) FFT parallèle sur CPU avec différents nombre de splits c) FFT parallèle sur Xeon-Phi avec différents nombre de splits

Chapitre 7 - Conclusion générale

La Transformée de Fourier Rapide (FFT – *Fast Fourier Transform*) est un élément clé dans de nombreuses applications et son implémentation sur les processeurs Multicœurs / Manycœurs a un grand intérêt. Nous la retrouvons depuis quelques années dans des applications clés telles que la communication sans fil à base de l'OFDM (*Orthogonal Frequency Division Multiplex*). Avec l'arrivée des réseaux C-RAN (*Cloud and Centralized Radio Access Network*) et de la migration des implémentations des traitements en bande de base de la couche physique des communication cellulaires vers le C-RAN, la mise en œuvre des FFTs sur les nœuds des multiprocesseurs devient une tâche non triviale.

Dans ce mémoire, nous nous sommes intéressés plus particulièrement à la communication sans fil à base du protocole LTE (*Long Term Evolution*) imposant une chaîne de traitement par lequel, plusieurs FFTs indépendantes doivent être calculées dans un intervalle de temps limité. Ce traitement nécessite le calcul de 100 FFTs indépendantes (avec des tailles de FFT allant de 128 à 2048 points) dans un temps de calcul inférieur à 66,7 µs afin de respecter les protocoles du LTE. Nous avons réalisé l'implémentation de transformées de Fourier rapide (FFT) sur des processeurs Multicœur et Manycœur. Une étude exhaustive a été présentée en ciblant trois plateformes différentes: un processeur Intel i7 (4 cœurs), un processeur Intel Xeon (8 cœurs) et un coprocesseur Intel Xeon Phi (61 cœurs). Nous avons pu montrer que des temps de calculs dépassaient le délai de 66,7 µs à

respecter. Il a donc été essentiel de proposer des solutions d'exécution de plusieurs FFT indépendantes et aussi la parallélisation de longue FFT.

Rappelons que l'objectif principal **de ce mémoire était de de** proposer des implémentations parallèles de FFTs indépendantes respectant les contraintes du protocole LTE basés sur l'OFDM ainsi que de longues FFT pour d'autres applications que les communications.

En utilisant Intel Math Kernel Library (MKL), dans notre approche de Xeon-Phi, nous avons réussi à réduire le temps d'exécution maximal de FFTs indépendantes. Nous avons proposé une implémentation sur les processeurs Multicœurs/Manycœurs en utilisant OpenMP pour réduire la latence du temps moyenne à 124 µs avec le mode *Native* après 1300 µs avec le mode *Offload*. Ces temps étant bien au-delà du délai à respecter du LTE. Ceci est un défi pour les projets exécutés sur des plateformes à mémoire partagée. Dans la première partie de ce travail, DPDK a montré que l'amélioration de performance a conçu seulement l'exécution d'une seule FFT. L'analyse statistique des FFTs indépendantes a donné des résultats qui ne respectent pas la norme LTE et dans ce cas on a remplacé l'outil par OpenMP. La deuxième partie de ce travail décrit comment ce niveau de performance peut être obtenu avec les plateformes Intel Core-i7, Xeon et le Manycœur Xeon-Phi. Les meilleurs résultats ont été obtenus avec le Xeon-Phi, qui a dépassé le Xeon Sandy-Bridge.

Pour exécuter cents FFTs dans une période de temps inférieure à 66.7 µs, une comparaison entre plusieurs méthodes d'implémentation a été étudiée. Les résultats expérimentaux montrent que la plateforme hétérogène Xeon + MIC avec MPI pure répond aux exigences de latence du LTE. Il a été montré que les paradigmes OpenMP et MPI fonctionnant uniquement sur les MICs ne peuvent pas profiter pleinement de la capacité de

calcul des architectures munies de nombreux cœurs. La combinaison hétérogène de Xeon + MICs a fourni une meilleure performance en tant que mélange des mémoires partagées et distribuées.

On a testé la robustesse de notre FFTs indépendantes, sur de longue FFT à paralléliser, d'où les trois étapes: division, traitement et recombinaison. Les résultats ont démontré un important gain de temps de calcul de 25 fois par rapport à FFTW (un thread) sur Xeon et de 20 fois plus rapide sur Xeon-Phi.

Bibliographie

[1] C. Ahn, S. Bang, H. Kim, S. Lee, J. Kim, S. Choi, J. Glossner; "Implementation of an SDR system using an MPI-based GPU cluster for WiMAX and LTE", Analog Integrated Circuit and Signal Processing, Springer, 2012, pp. 569-582

[2] Y. Bouguen, É. Hardouin, F. Wolff, "LTE et les réseaux 4G", Eyrolles, 2012

[3] Y. Soo Cho, "MIMO-OFDM Wireless Communications With Matlab", John Wiley& Sons, 2010, (chapters 4 and 5)

[4] H. Schulze, C. Luders, "Theory and Applications of OFDM and CDMA", Wiley, 1st edition, 2005

[5] C. Cox, "An Introduction To LTE", Wiley, 1st edition, 2012

[6] S. Schwarz, J. Colomikuno, M. IMKO, M. Taranetz,Q. Wang, M. Rupp, "Pushing the Limits of LTE: A Survey on Research Enhancing the Standard", IEEE , May 2013, pp. 51-62

[7] G. Boudreau, J. Panick, N. Guo, R. Chang, N. Wang, S. Vrzic, "Release8/Interference Coordination and Cancellation for 4G Networks", IEEE Communications Magazine, April 2009, pp.74-81

[8] K. Ravindhra, S. Subramanian, U. Sankar, "Long Term Evolution Downlink Physical Layer Simulation in Matlab and Simulink", International Journal of Future Computer and Communication, Vol. 1, No. 2, August 2012, pp. 131-134

[9] L. Zhou, H. Zhiqiang, D. Ran, H. Lifeng, "An Implementation of MIMO Detection in TD-LTE based on General Purpose Processor", ICST International conference on Communications and Networking, China, Aug 2012, pp. 807-811

[10] P. Emmerich, D.Raumer, F.Wohlfart, G.Carle, "A Study of Network Stack Latency for Game Servers", 13th Annual Workshop on Network and Systems Support for Games (NetGames), Nagoya, December 2014, pp. 1-6

[11] T.Babette, C.Soladani, L.Mathy, "Fast Userspace Packet Processing", ACM/IEEE Symposium Architectures for Networking and Communications Systems (ANCS), Oakland CA, May 2015, pp. 5-16

[12] H.Wippel, "DPDK-based Implementation of Application-tailored Networks on End User Nodes", International Conference and Workshop on the Network of the Future (NOF), Paris, December 2014, pp.1-5

[13] H. Zhou, X. Wang, "A Parallel Computing Component for Linux Cluster with Threads Binding Supports", Computational Intelligence and Software Engineering (CiSE), Wuhan, December 2010, pp. 1-4

[14] P. Rodriguez, "A radix-2 FFT algorithm for modern single instruction multiple data (SIMD) Architectures", IEEE Acoustics, Speech, and Signal Processing (ICASSP), Florida USA, May 2002, pp. 3220-3223

[15] G.Locharla, S.Kumar, K Mahapatra, S.Ari,"Implementation of Input Data Buffering and Scheduling Methodology for 8 parallel MDC FFT", Int. Symposium on VLSI Design and Test, India, June 2015, pp.1-6

[16] R. Mego, T. Fryza, "Performance of parallel algorithms using OpenMP", Conf. Radioelektronika, Czech Republic, April 2013, pp. 236-239

[17] H. Zhou, X. Wang, "A Parallel Computing Component for Linux Cluster with Threads Binding Supports", Computational Intelligence and Software Engineering, Wuhan, December 2010, pp. 1-4

[18] K. G.Lenzi, F.A.P. Figueiredo, J.A.B. Filho, "Fully Optimized Code Block Segmentation Algorithm for LTE-Advanced", Wireless Comm. and Networking Conf., Shanghai, 2013, pp. 3312-3317

[19] M. Khelifi, D. Massicotte, Y. Savaria, "Parallel Independent FFTs Implementation on Intel Processors and Xeon Phi for OFDM and LTE systems," Conference on Nordic Circuits and Systems (NORCAS), Oslo, Oct. 2015, pp. 1-4

[20] H. Kamal, A. Wagner, "FG-MPI: Fine-grain MPI for multicore and clusters," Int. Symp. on Parallel & Distributed Processing, Workshops and Phd Forum, Atlanta, April 2010, pp.1-8

[21] Y. Li, W. Shen, A. Shi, L. He, D. Zhao, "MPI and OpenMP Paradigms on Cluster with multicores and its application on FFT", Int. Conf. on Computer Design and Application, Qinhuangdao, June 2010, pp.V1-23 – V1-26

[22] M. D.Jones, R. Yao "Parallel programming for OSEM reconstruction with MPI, OpenMP, and hybrid MPI-OpenMP," Nuclear Symposium Conference Record, Rome, Oct. 2004, pp.3036-3042

[23] P.D'Ambra, M.Danelutto, D.di Serafino, M.Lapegna, "Integrating MPI-based numerical software into an advanced parallel computing environment," Conf. on Parallel, Distributed and Network-based Processing, Italy , Feb. 2003, pp.283-291

[24] K. G.Lenzi, F.A.P. Figueiredo, J.A.B. Filho, "Fully Optimized Code Block Segmentation Algorithm for LTE-Advanced," Wireless Comm. and Networking Conf., Shanghai, 2013, pp. 3312-3317

[25] M.Fallahpour, C.Lin, M.Lin, C.Chang, "Parallel One- and Two-Dimensional FFTs on GPGPUs", International Conference on Anti-Counterfeiting, Security and Identification, Taipei, 2012, pp.1-5

[26] A.Mirza, "Parallel Computation of the Interleaved Fast Fourier Transform With MPI", M.Sc Thesis, University of Akron, Ohio, December 2008

[27] J.Wiggs, H.Jonsson"A Hybrid Decomposition Parallel Implementation of the Car-Parrinello Method", Computer Physics Communication (in Press), University of Washington, Seattle, Feb 2008

[28] S.Goedecker, M.Boulet, T.Deutsch, "An efficient 3-dim FFT for plane wave electronic structure calculations on massively parallel machines composed of multiprocessor nodes", Elsevier, Computer Physics Communications 154, March. 2003, pp 105–110

[29] T.Hoefler, S.Gottlieb, "Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes", Int Conf on Anti-Counterfeiting, Security and Identification, Taipei, 2012, pp.1-5 17th European MPI users group meeting conference on Recent advances in the message passing interface, Berlin, 2010, pp. 132-141

[30] S.Meiyappan, "Implementation and performance evaluation of parallel FFT algorithms", National University of Singapore, School of computing, Singaore

[31] M.Pippig,"PFFT: An Extension of FFTW to Massively Parallel Architectures", Society for Industrial and Applied Mathematics (SIAM J. SCI. COMPUT), 2013, Vol. 35, No. 3, pp. C213–C236

[32] E.Wang, Q.Zhang, G. Zhang, X.Lu, Q.Wu, "High Performance Computing on Intel Xeon Phi," Springer Inc., 2012 (chapters 1 to 6 and 11)

[33] Intel corporation, "Intel Math Kernel Library",Reference Manual, August, 2008,http://fulla.fnal.gov/intel/mkl/mklman.pdf (Visited 1st February 2016)

[34] S. Qi, J. Feng, L.Xiao, "Single and multi-threading MRI reconstruction based on FFTW 2.X and Intel MKL", Computer Science and Automation Engineering (CSAE), Shanghai, June 2011, pp. 124-127

[35] I. Cerrato, M. Annarumma, F. Risso, "Supporting Fine-Grained Network Functions through Intel DPDK", Third European Workshop on Software Defined Networks (EWSDN), Budapest, September 2014, pp. 1-6

[36] G. Pongracz, L. Molnar, Z. Kis, "Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK", Second European Workshop on Software Defined Networks (EWSDN), Berlin, October 2013, pp. 62-67

[37] R. Mego, T. Fryza, "Performance of parallel algorithms using OpenMP", Conf. Radioelektronika, Czech Republic, April 2013, pp. 236-239

[38] J. Jeffers, "Intel Xeon Phi Coprocessor High Performance Programming", Elsevier Inc., 2013 (chapters 2 and 3)

[39] S.B.Chappell and A.Stokes "Parallel Programming with Intel Parallel Studio XE", J. Wiley & Sons, 2012 (chapters 4-9)

[40] N. Hinitt, T. Kocak, "GPU-Based FFT Computation for Multi-GigabitWirelessHD Baseband Processing", EURASIP J. on Wireless Comm. and Networking, Vol. 2010, 13 pages

[41] T. Taleb, "Toward Carrier Cloud: Potential, Challenges, and Solutions", IEEE Wireless Communications, June 2014, pp. 80-91

[42] D. Wübben, P. Rost, J. Bartelt, M. Lalam, V. Savin, M. Gorgoglione, A. Dekorsy, and G. Fettweis, "Benefits and Impact of Cloud Computing on 5G Signal Processing - Flexible centralization through cloud-RAN", IEEE Signal Processing Magazine, Nov. 2014, pp. 35-44

[43] D. Sabella, A. De Domenico, E. Katranaras, M.A. Imran, M. Di Girolamo, U. Salim, M. Lalam, K. Samdanis, A. Maeder, "Energy Efficiency Benefits of RAN-as-a-Service Concept for a Cloud-Based 5G Mobile Network Infrastructure", IEEE Access Journal, Vol. 2, Dec. 2014

[44] M. Khelifi, D. Massicotte, Y. Savaria, "Towards Efficient and Concurrent FFTs Implementation on Intel Xeon/MIC cluster for LTE and HPC Applications", IEEE Int. Symp. on Circuit and Systems (ISCAS), Montreal, Mai 2016

[45] M. Frigo, and S. G. Johnson, "The Design and Implementation of FFTW3" Proceedings of the IEEE, vol. 93, no. 2, February 2005, pp. 216–231

[46] J.W. Cooley, J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", Mathematical Computer 19, pp. 297-301, April 1965

Annexe A – Article publié – NORCAS 2015

M. Khelifi, D. Massicotte, Y. Savaria, "Parallel Independent FFTs Implementation on Intel Processors and Xeon Phi for LTE and OFDM Systems", IEEE Nordic Circuits and Systems Conference (NORCAS), Oslo, Norway, Oct. 2015.

Parallel Independent FFT Implementation on Intel Processors and Xeon Phi for LTE and OFDM Systems

Mounir KHELIFI[1], Daniel MASSICOTTE[1] and Yvon SAVARIA[2]

[1]Université du Québec à Trois-Rivières, Electrical and Computer Engineering Department, Trois-Rivières, Quebec, Canada
Laboratoire des Signaux et Systèmes Intégrés, Groupe de recherche en électronique industrielle
[2]École Polytechnique de Montréal, Electrical and Computer Engineering Department, Montréal, Quebec, Canada
{mounir.khelifi, daniel.massicotte}@uqtr.ca, yvon.savaria@polymtl.ca

***Abstract*— Fast Fourier Transform (FFT) is a key element for wireless applications based on the OFDM (Orthogonal Frequency Division Multiplexing) and challenging for implementing on processor multi-cores/many-cores. As an example, the Long Term Evolution (LTE) protocol establishes a requirement for processing, whereby many independent FFTs must be calculated within a limited time slot. By using Intel Math Kernel Library (MKL), in our approach to Xeon phi, we managed to reduce the maximum execution time of many independent FFTs. We proposed an implementation on processors multi-cores/many-cores using OpenMP (Open Multi-processing) reducing the mean time latency to 124 µs on native mode after 1300 µs with the offload. This is a challenge for shared memory projects. This paper describes how this level of performance can be obtained with multi-core Intel i7, Xeon processors and a many-core Xeon Phi. The best results were obtained with the Xeon Phi, which outperformed the Xeon Sandy-Bridge.**

Keywords — LTE, OFDM, Fast Fourier Transform (FFT), Parallel programming, multithread, Parallel, multi-core, many-core, MKL

I. INTRODUCTION

Long Term Evolution (LTE) aims to provide an increased data rate and reduced transmission delays compared with older wireless telecommunication standards. However, its complex protocol stack imposes stringent constraints on the physical layers, making the implementation of compute-intensive operations a challenge [6]. LTE signal processing relies heavily on channel decoding, channel estimation, Fast Fourier Transform (FFT) processing, and other processing blocks. All these physical layer processings must be accomplished within the time slot. At the same time, the packet-based infrastructure imposes very short data transmission latencies at the physical (PHY) and media access control (MAC) layers [5]. For instance, the smallest scheduling interval in LTE is currently 1 ms; up to 100 subcarriers must be processed during this time, which imposes a processing time limit of 66.7 µs. Future 60 GHz protocols, such as WirelessHD, call for lowering this processing time to under 200 ns [6][10]. Thus, real-time requirements are serious obstacles to equipment infrastructures intended to deploy soft-switches, virtual machines and software-defined network technologies.

This paper investigates the use of processors multi-cores/many-cores to compute many FFTs of various sizes as specified in Release 8 of the LTE standard for all specified channel bandwidths (1.25 MHz to 20 MHz), FFT sizes (128 to 2048 points) and sampling frequencies (1.92 MHz to 30.72 MHz) [5][10]. As shown in Table 1, the LTE physical layer used a number of Resource Blocks (RB) depends of the channel bandwidth [10]. Thus, considering that for each RB, one FFT is executed, for the case of a 20 MHz channel bandwidth, 100 RBs are used, involving 100 FFTs of size 2048 to be executed concurrently. FFT computation within LTE will be parallelized into multi-core core-i7, Xeon and many-core Xeon Phi in order to improve computational time performance.

978-1-4673-6576-5/15/$31.00 ©2015 IEEE

The highlighted parallelization techniques found in the literature examine the implementation of eight parallel FFTs suitable for MIMO-OFDM using eight parallel streams operated in variable lengths of 64/128/256/512 [1]. Furthermore, the authors demonstrate how the increase of the number of threads and the size of input FFT data influence the performance compared to the sequential multithreaded mode on core-i7 and Xeon with OpenMP [3]. In addition, since the FFT application based on Linux servers uses an Intel Xeon E5506 processor and clusters, the experiments indicate that the element with thread binding back up can be utilized for high performance computing (HPC) tasks [2]. OFDM suffers from a high processing in physical layer and the latency is a critical constrain challenging the execution of massively concurrent FFTs. Compared to FPGAs or ASICs technologies, the processor multi-core implementation for centralized radio access network (C-RAN) offers a reconfigurable platform [13]. LTE can adopt the many-core Xeon Phi as a substitute to the high-cost FPGA and the single-core DSP, which require a low-level complex programming unlike the easy C/C++ standard programming for Xeon Phi.

Our contribution uses all the following techniques to implement 100 independent FFTs on a Xeon Phi cluster for the OFDM/LTE physical layer. By using Native and Offload modes for Xeon Phi FFT implementation, an order of magnitude computation speedup could be achieved on the already parallelized code. This includes different scenarios of memory size, bandwidth configuration and a number of Xeon Phi. In addition, we will compare the results with those of a multi-core Central Processing Unit. This should lead to valuable knowledge about processing time, power consumption, hardware complexity, and the efficiency of each platform.

Table 1 LTE Uplink physical layer parameters

Channel Bandwidth (MHz)	1.25	2.5	5	10	15	20
FFT size	128	256	512	1024	1536	2048
Number of RB	6	12	25	50	75	100

In Section II, we explain the multiple FFTs, the thread affinity concept and to way they affect the proposed parallel model. Section III, highlight memory allocation issues and multithreading in the application. Section IV describes the different implementation modes on Xeon Phi. Section V discuss the results, and Section VI presents the conclusion.

II. MULTIPLE FFTS AND THE THREADS AFFINITY

Our paper introduces the parallelization technique on Parallel Studio XE 2013 with the use of the OpenMP library. We explain the difference between the two types of FFTs we tested: Sequential-Multithreaded and multiple Parallel FFTs. For the

sequential multithreaded mode, we performed only one FFT by requesting the compiler to use all the threads randomly for the execution. For the parallel independent FFTs, we performed multiple FFTs on many threads by controlling the affinity with OpenMP [3]. The OpenMP library accelerates the FFTs calculation thanks to the cores affinity KMP_AFFINITY [11], which helps in control the number of threads per core to use in the application. In fact, this depends on the topology of the machine, that is, whether there are two threads per core, or four threads per core, and the processing speed and the memory type are "distributed or shared". The thread affinity has a positive influence on the application's performance. There are two important factors for OpenMP affinity: first, it dictates the number of threads to use and second, it recognizes the Thread ID of all the available threads [11]. The code program using the OpenMP begins its execution in a sequential mode using only a single thread until it encounters the pragma switch *"#pragma omp parallel"*. Inside this region, the sequential thread becomes the master, and a memory pool is created as well. In this enclosed pragma statement, all the independent FFTs within will be executed in parallel [3]. At the end of the parallel execution, the first thread in the team to arrive waits for all fellow threads to complete their tasks, upon which the created team is dissolved and we return to the sequential execution (Figure 1). In our application, only the execution time of the OpenMP parallel region is measured.

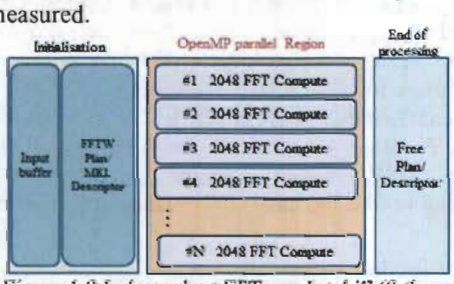


Figure 1 8 Independent FFTs on Intel i7 (8 threads).

On the 8 thread Intel core-i7, we begin by reclaiming the number of threads available on the platform, which is 8; each thread has its own identifying "ID" number, which ranges from 0 to 7. When the data are available for the threads to execute, the order of the IDs will be random. When we start a parallel region, the procedure is comparable to a horse race where the horses are the threads running in random directions [3]. The FFT tasks should be proportional to the thread's ID number. Where an FFT [ID]=&array_temp[ID*length], the 8 independent parallel FFTs run together as follows:

*FFT0 = &array_temp[0*2048]*
...
*FFT7 = &array_temp[7*2048]*

When ID 0 finishes up, its data have already been written into a buffer. During this process, IDs 1 to 7 continue working. When ID 1 finishes its task, the data writing for ID 0 is completed. The same holds for ID 2, which finishes its task when the ID 1 writing is over, and so on and so forth [9]. The same method is used with the Xeon Sandy-Bridge processor and the Xeon Phi coprocessor.

III. Multithreading and Memory Sharing

We used two different affinity techniques to control the threads. The first affinity is Compact which maintains all threads processing on a single processor. This is when all the threads in the program must enter the different parts of a large array multiple times. All the cores on the same processor enter that processor's memory banks [7]. When our threads are to access the data saved in a single processor's memory, it is advisable to place them on the processor that hosts that memory zone. The second affinity is the Scatter or round-robin. This affinity tests whether the threads are independent and do not access several memory zones contrary to the previous Compact mode [7]. The advantage of this affinity is that not all the threads have to access and share the same memory sections and cache. As a result, the memory latency becomes higher when the threads must begin entering memory zones that can be owned by another processor.

For Xeon Phi, there are two obvious methods of handling the non-shared memory between a Xeon Phi coprocessor and the Xeon processor. The first is data organization, where the compiler runtime creates and mails out data buffers to coprocessors by organizing the parameters supplied by the application. The second method is the virtual shared memory model, which depends on system level runtime support to preserve data consistency between the Xeon's and Xeon Phi's virtual shared memory address space [11]. The first method is supplied by OpenMP spec and is more common than the second. It is the method employed in this work.

IV. The Implementation Modes on Xeon Phi

Unlike Xeon, which has 2 threads per core, Xeon Phi owns 4 threads per core. Before compiling the code, it is important to set the environmental variables OMP_NUM_THREADS and KMP_AFFINITY. Choosing the Compact affinity will set OpenMP threads by allocating cores one by one, whereas choosing the Scatter affinity will place one thread in every core until the total number of cores is swept. We ran an application on the Xeon Phi coprocessor using two modes; Native and Offload.

Native execution is when the program runs completely on the Xeon Phi. In this Native mode, we can use an existing code with negligible changes. The simplest mode of running applications on the Xeon Phi coprocessor is the Native. The program is compiled on the host using the compiler key -mmic to create a code for the (Many Integrated Core) MIC architecture. The binary can then be transferred to the Xeon Phi and has to be begun there. The Xeon Phi coprocessor has its own Linux operating system and IP address. As a result, the Xeon Phi support on-card operation mode for copying programs and variables onto the Xeon Phi card manually and then running the programs directly on the Xeon Phi card. There are many benefits to running directly on the MIC: time usually used for data transmission is not wasted, and the same data can be processed as often as desired [11]. To copy libraries from the host and execute the program on the MIC we used "micnativeloadex" during the build.

```
MIC_ENV_PREFIX=PHI
MIC_ NUM_THREADS=100
PHI_KMP_AFFINITY='Compact/Scatter'
icc -mmic -mkl -openmp fftmklpara.cpp -o parallel.MIC
micnativeloadex parallel.MIC -e "OMP_NUM_THREADS=100"
```

The second implementation mode on Xeon Phi is the Offload. OpenMP pragmas can be added to C/C++ programs to stamp areas of code that should be offloaded to the MIC and run there. This approach is very similar to the accelerator pragmas introduced by the PGI compiler and OpenACC to offload programs to General Purpose GPU (GPGPUs) [7]. When the

Intel compiler encounters an offload pragma, it creates code for both the Xeon Phi and the Xeon. The code to copy the data to the MIC is automatically generated by the compiler, but we can improve the data transfer by inserting data clauses into the Offload pragma [11]. The disadvantage of the Offload mode is the offloading time that decreases the application's performance.

V. Evaluation of Performance and Results

A. Simulation environment and Platforms

The first platform consists of the Intel i7-4770, with a frequency of 3.6 GHz, 16 GB DDR4 of RAM, 20 MB L3 of cache memory, 4 cores and 8 threads for ultimate multitasking. The operating system is Ubuntu 13.04 with a Kernel version 3.8.8. The second platform is the Intel Xeon CPU E5-2650, with a frequency of 2.0 GHz, 64 GB DDR4 of RAM, 20 MB L3 of cache memory, 8 cores and 16 threads. The operating system is CentOS 6.5 with a Kernel version 2.6.3. The third platform, from Calcul Quebec, is the Xeon Phi coprocessor (many-core) that contains 61 cores and 244 threads. Each core runs with the frequency of 1 GHz and has an L2 cache memory size of 512 KB that interacts with the other cache memories of the other cores. The cores are interconnected by a high speed bidirectional ring. The coprocessor (Slave) connects with the Xeon processor (Master) using the PCI bus to exchange data [4]. We have two coprocessors per host that represent the central processing Unit.

The programming language is C/C++ and the main compiler used is Intel. The MKL and OpenMP libraries are located in the parallel studio XE 2013. The tasks were executed in both the sequential and parallel modes. In each experiment, 10^6 FFTs of complex float number of size N are executed on the three platforms. To analysis the distribution of 10^6 computational times, we considered the statistical metrics to show the method performance based on three key characteristics [12]: central tendency, dispersion and the shape. The moments of distributions are define by the first moment (mean), second moment (coefficient of variation – CV) and the third and fourth moments (skewness – α and kurtosis – κ). The last one gives more information to measure the outliers prone distribution around the mean and to reach a deterministic computation time.

B. Simulation Results on Parallel Studio XE 2013

The 1×SMFFT is a sequential multithreaded FFT that uses all the platform's threads and where the compiler allocates threads randomly. The proposed method in this work introduces the P×PMFFT which presents parallel independent P FFTs running on P threads.We used the following method names in the results:

- 1×SMFFT – one sequential multithreaded FFT
- P×PMFFT– P parallel multithreaded FFTs

The ideal results a predictable computation time. Tables 2 to 4 show the statistical metrics. Low kurtosis and skewness values present a more predictable computation time and a Gaussian distribution close to mean value [12].

Table 2 Computation time of FFTs on Intel i7-4770

Methods	Affinity	Min (µs)	Max (µs)	Mean (µs)	CV	α	κ
1×SMFFT MKL	Without	7	43	7.9	0.044	8.9	471
8×PMFFT MKL	Scatter	14	100	15.3	0.097	26.8	1044
8×PMFFT MKL	Compact	14	150	15.2	0.104	32.4	1644

The performance on Intel i7 is shown in Table 2. One sequential multithreaded FFT generates many outliers of computaional time and a high kurtosis. In the parallel mode, we observe that the MKL FFT has a higher maximum with the Compact affinity due to the false sharing errors. For i7, we note that the computation time is ×4 faster in parallel than sequential. With Xeon, Table 3 and Figure 2 show the metrics and boxplot, respectively. We observe that MKL performs well for parallel independent implementation. For Intel i7 and Xeon, we observe a gain in the mean computation time when the number of parallel FFTs increases. The computational time of 1×SMFFT with OpenMP directives is slow, even if the maximum number of threads is set to 8 or 16. In fact, the threads communicate with each other and access the same memory, because inputs and outputs are defined as shared variables [3]. That is the disadvantage of shared memory.

Table 3 Computation time of FFTs on Intel Xeon for N=2048

Methods	Affinity	Min (µs)	Max (µs)	Mean (µs)	CV	α	κ
1×SMFFT MKL	Without	16	96	26.9	0.238	5.3	36.3
16×PMFFT MKL	Scatter	16	77	18.7	0.291	6.8	52.9
16×PMFFT MKL	Compact	16	42	17.9	0.101	6.3	59.1

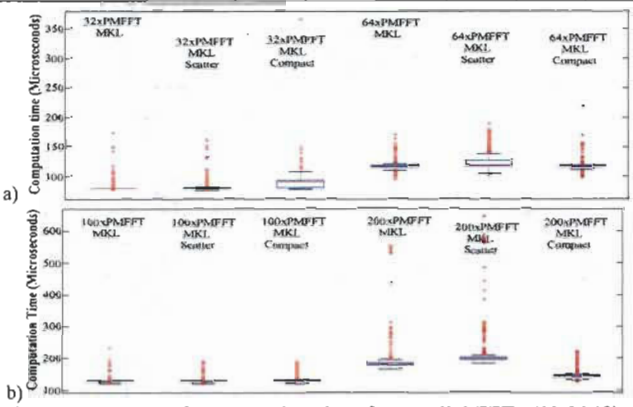


Figure 2 Boxplot of computation time for parallel FFTs (N=2048) on Xeon Phi for a) 32 and 64 and b) 100 and 200 parallel FFTs.

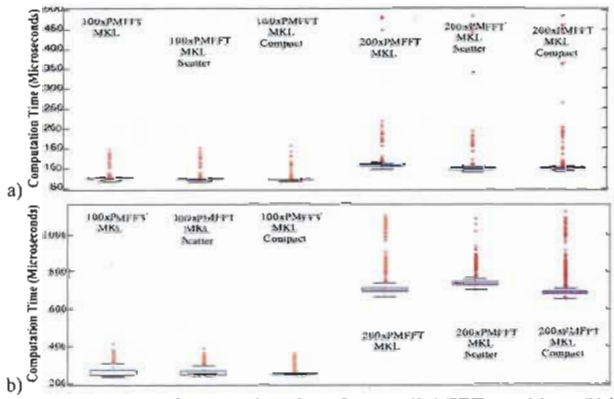


Figure 3 Boxplot of compution time for parallel FFTs on Xeon Phi for 100 and 200 parallel FFTs with a) N=1024 and b) N=4096.

The Xeon Phi provided better statistical analysis results for a large number of independent FFTs. According to Table 4 and Figure 3a, when the number of parallel FFTs increases, Compact affinity performs better than Scatter affinity. The mean time of 200×SMFFT MKL on Intel Xeon is 5400 µs and 200×PMFFT MKL on Xeon Phi with Compact affinity is 140 µs, effectively giving us a time speedup of 38. The speedup computation time rises with the number of parallel FFTs and the FFT size. The good scaling capabilities of Xeon Phi encouraged us to double FFT input size to 4096 (Figure 3b); here again, results were uniform with low skewness and kurtosis values.

Table 4 Computation time on Intel Xeon Phi for P×PMFFT MKL method to execute P FFTs in parallel for 1024, 2048 and 4096 FFT size

Number of parallel FFTs	Affinity	Min (µs)	Max (µs)	Mean (µs)	CV	α	κ
			N=1024				
100×PMFFT MKL	Scatter	67	152	72.0	0.091	5.5	49.1
	Compact	67	156	69.9	0.060	12.6	198
200×PMFFT MKL	Scatter	91	483	98.5	0.185	16.5	310
	Compact	92	482	99.9	0.227	13.6	204
			N=2048				
32×PMFFT MKL	Without	76	172	78.9	0.063	8.7	116
	Scatter	76	160	78.9	0.069	7.4	80.9
	Compact	78	365	87.2	0.103	15.2	458
64×PMFFT MKL	Without	95	170	115	0.048	2.0	21.8
	Scatter	100	189	122	0.100	1.3	4.7
	Compact	97	216	115	0.050	4.5	65.6
100×PMFFT MKL	Without	119	230	124	0.048	8.5	98.2
	Scatter	118	190	124	0.043	7.9	78.1
	Compact	119	188	128	0.078	2.3	9.0
200×PMFFT MKL	Without	166	552	182	0.151	10.7	133
	Scatter	184	644	200	0.159	10.2	117
	Compact	124	220	140	0.056	5.0	38.6
			N=4096				
100×PMFFT MKL	Without	234	409	255	0.105	1.6	4.5
	Scatter	236	385	256	0.099	1.6	4.4
	Compact	236	359	243	0.054	5.6	36.9
200×PMFFT MKL	Without	667	1103	710	0.073	4.4	25.9
	Scatter	694	1084	740	0.045	3.4	19.5
	Compact	653	1121	694	0.071	4.8	31.0

Figure 4 shows the difference between the Native and Offload modes on Xeon Phi for 100 and 200 FFTs using the Compact affinity. To observe the trend, we explore the FFT size beyond the LTE standard. This figure reveals the mean, the maximum and the CV value for different FFT sizes ranging from 64 to 16384. Noted that all results are in logarithm scale. On one hand, it is clear the Native mode is better, since it has a lower maximum and mean computation time for all the different FFT sizes, measure of dispersion seems to be better in Native mode. One the other hand, the more we increase the FFT size, more the difference between the methods reduced.

VI. Conclusion

In this paper, we investigated the use of processors multi-cores/many-cores to compute parallel independent FFTs for OFDM used in LTE systems. The statistical metrics show that executing large FFT sizes implemented on many-core platforms is far better than implementing small FFT size on processor multi-core. This paper is a contribution to the LTE implementation on processor multi-core/many-core for future centralized radio access network.

Acknowledgment

The authors wish to express their thanks to the ReSMiQ and CRSNG for their financial support, Calcul Quebec for HPC equipment and their dedicated help, specifically Dr. Bart Oldeman from McGill HPC team.

References

[1] G.Locharla, S.Kumar, K Mahapatra, S.Ari,"*Implementation of Input Data Buffering and Scheduling Methodology for 8 parallel MDC FFT*", Int. Symp. on VLSI Design and Test, India,June 2015, pp.1-6

[2] H. Zhou, X. Wang, "*A Parallel Computing Component for Linux Cluster with Threads Binding Supports*", Computational Intelligence and Software Engineering, Wuhan, Dec. 2010, pp. 1-4

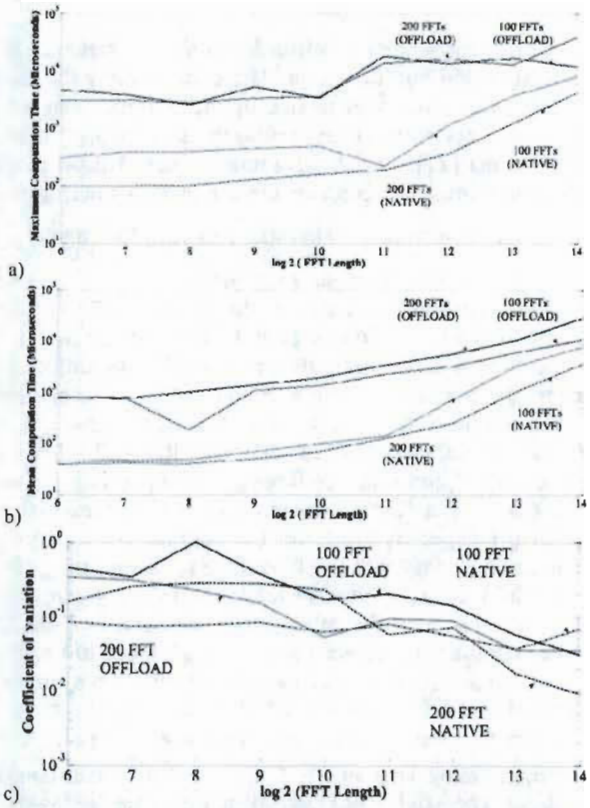


Figure 4 Comparaison plots between the two modes on Xeon Phi (Native and Offload) for 100 and 200 parallel FFTs using Compact affinity: maximum (a) and mean (b) computation time and CV (c).

[3] R. Mego, T. Fryza, "Performance of parallel algorithms using OpenMP", Conf. Radioelektronika, Czech Republic, April 2013, pp. 236-239

[4] P. Rodriguez, "*A radix-2 FFT algorithm for modern single instruction multiple data (SIMD) Architectures*", Int. Conf. on Acoustics, Speech, and Signal Processing (ICASSP), Florida, May 2002, pp. 3220-3223

[5] S.Bhattacharjee, S.Singh, S.Patra, "*LTE Physical Layer Implementation Using GPU Based High Performance Computing*", IEEE Int. Conf. on Advanced Comm. Control and Computing Technologies, Ramanthapuran, May 2014, pp. 1546-1550

[6] S. Schwarz et al., "*Pushing the Limits of LTE: A Survey on Research Enhancing the Standard*" Digital Object Identifier, Access IEEE (Volume 1), 2013, pp. 51-62

[7] N. Hinitt, T. Kocak, "*GPU-Based FFT Computation for Multi-GigabitWirelessHD Baseband Processing*", EURASIP J. on Wireless Comm. and Networking, Vol. 2010, 13 pages.

[8] S. Qi, J. Feng, L.Xiao, "*Single and multi-threading MRI reconstruction based on FFTW 2.X and Intel MKL*", IEEE Int. Conf. on Computer Science and Automation Engineering (CSAE), Shanghai China, 10-12 June 2011, pp. 124-127.

[9] S.B.Chappell and A.Stokes "*Parallel Programming with Intel Parallel Studio XE*", J. Wiley & Sons, 2012 (chapters 4-9)

[10] Y. Soo Cho "*MIMO-OFDM Wireless Communications With Matlab*", John Wiley& Sons, 2010 (chapters 4 and 5)

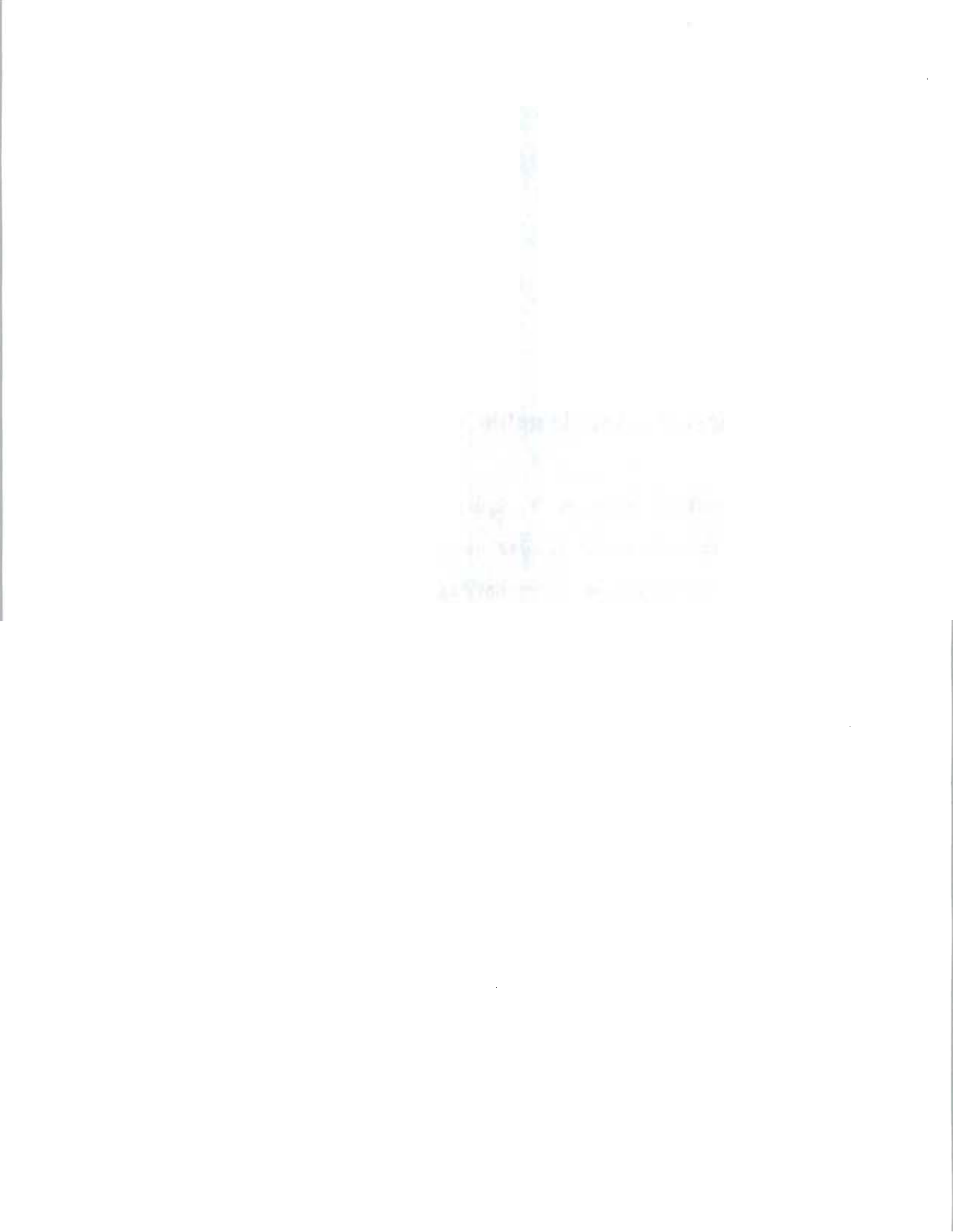
[11] J. Jeffers, "*Intel Xeon Phi Coprocessor High Performance Programming*", Elsevier Inc., 2013 (chapters 2 and 3)

[12] L.T. DeCarlo, "*On the Meaning and Use of Kurtosis*", J. Psychological Methods, American Psychological Association, 1997, pp. 292-307

[13] K. G.Lenzi, F.A.P. Figueiredo, J.A.B. Filho, "*Fully Optimized Code Block Segmentation Algorithm for LTE-Advanced*", Wireless Comm. and Networking Conf., Shanghai, 2013, pp. 3312-3317

Annexe B – Article publié – ISCAS 2016

M. Khelifi, D. Massicotte, Y. Savaria, "Towards Efficient and Concurrent FFTs Implementation on Intel Xeon/MIC cluster for LTE and HPC", IEEE International Symposium on Circuit and Systems (ISCAS), Montreal, Mai 2016.



Towards Efficient and Concurrent FFTs Implementation on Intel Xeon/MIC Clusters for LTE and HPC

Mounir KHELIFI[1], Daniel MASSICOTTE[1] and Yvon SAVARIA[2]

[1]Université du Québec à Trois-Rivières, Electrical and Computer Engineering Department, Trois-Rivières, Québec, Canada
Laboratoire des Signaux et Systèmes Intégrés, Groupe de recherche en électronique industrielle
[2]École Polytechnique de Montréal, Electrical Engineering Department, Montréal, Quebec, Canada
{mounir.khelifi, daniel.massicotte}@uqtr.ca, yvon.savaria@polymtl.ca

***Abstract*—Fast Fourier Transform (FFT) is an important part of many applications, such as in wireless communication based on OFDM (Orthogonal Frequency Division Multiplexing). With Cloud Radio Access Networks, implementing FFTs on multiprocessor clusters is a challenging task. For instance, supporting the Long Term Evolution (LTE) protocol requires processing 100 independent FFTs (with sizes ranging from 128 to 2048 points) in 66.7 µs. In this work, seven native FFT candidate implementations are compared. The considered implementation environments are: OpenMP (Open Multi-Processing) on 1 core, MPI (Message Passing Interface) on 1 core, 2 cores, and 3 cores, Hybrid OpenMP+MPI on 1 core and 3 cores, and MPI on an heterogeneous platform composed of Xeon-Phi and 3 cores. The reported experimental results show that the latter method meets the latency requirements of LTE. It is shown that the OpenMP and MPI paradigms running only on MICs (Many Integrated Cores) cannot benefit fully from the computing capability of many-core architectures. The heterogeneous combination of Xeon+MICs provides a better performance.**

Keywords — LTE, OFDM, FFT, Parallel programming, Many Integrated Core (MIC), MKL, OpenMP, MPI, Hybrid, Many-core, Multi-core

I. INTRODUCTION

With the increasing prevalence of parallel platforms and software tools, programmers face the challenge of choosing the most adapted architectures. Many techniques have been proposed to leverage parallelism while taking advantage of both shared and distributed memories.

This paper reports on the use of the Xeon-Phi many-core and the more mainstream Xeon multi-core CPU to compute numerous independent FFTs, all with the same size and for various sizes, as specified in Release 8 of the LTE standard, for all specified channel bandwidths (1.25 MHz to 20 MHz) and FFT sizes (ranging from 128 to 2048 points) [1] [3]. The LTE physical layer use a number of so-called Resource Blocks (RBs) depending on the channel bandwidth [1]. Thus, considering that one FFT is executed in each RB, in the case of a 20 MHz channel bandwidth, 100 RBs are used, involving 100 FFTs of size 2048 to be executed concurrently. This paper investigates the best means to perform these 100 parallel FFT computations in real time as required by LTE. It will explore how they can be parallelized and executed on Xeon/Xeon-Phi (MIC – Many Integrated Core) in order to improve the processing time.

It was found that OpenMP can suffer from memory leaks and execution time slow down [2], and it has a limited scalability as when executing 100 FFTs, the processing time can reach 180 µs, while a mean value of 120 µs was observed on an MIC system [3]. The implementation of MPICH2 for fibers allows the execution of hundreds to thousands of MPI processes without the need for so many cores [4]. In addition, when multiple FFTs are executed on Linux servers running over Intel Xeon E5506 processors and clusters, experiments indicate that systems with thread binding back up can be utilized for high performance computing (HPC) tasks [5]. An implementation performing eight parallel FFTs of size 64/128/256/512 using eight parallel processing streams resulted in good performance [6]. To confirm the efficiency of hybrid OpenMP+MPI implementations, MPI was used for internode communication outside the parallel region, and OpenMP was used inside the nodes [7]. In a comparative study, applications were implemented using: 1) MPI coordinating tasks over a large Linux distributed memory SMP (Shared Memory multi-Processor) cluster, 2) OpenMP running over a distributed/shared memory system within the Altix 3700 platform, and on a shared memory Dell PowerEdge 6650, and 3) a hybrid combination of MPI and OpenMP implemented on the same clusters [8]. Best speed up was obtained with pure MPI running over a Shared Memory Multiprocessor (SMP) cluster [8]. A statistical analysis of the performance obtained when encapsulating the Fastest Fourier Transform in the West (FFTW) in parallel computing environments was performed [9]. Experiments showed that efficiency improves with the number of nodes [9]. With LTE processing, latency is a critical constraint when performing many concurrent FFTs. Compared to FPGA or ASIC technologies, a many-core implementation supporting cloud radio access network (C-RAN) offers a great deal of flexibility [10]. This paper explores using Xeon and Xeon-Phi as a substitute to high-end FPGAs and single-core DSPs that require low-level complex programming. By contrast C/C++ standard programming is used with Xeon and Xeon-Phi.

The rest of this paper compares several means of implementing 100 independent FFTs in parallel. The selected target performance constraint is set to 66.7 µs, inspired by the OFDM/LTE physical layer. The following techniques and architectures are considered: Xeon and Xeon-Phi clusters with OpenMP, MPI and a Hybrid OpenMP+MPI.

In Section II, the different available implementation modes on Xeon-Phi combined with the OpenMP and MPI software tools are reviewed. Section III proposes a hybridization of OpenMP and MPI. Section IV reports the obtained results and Section V summarizes the conclusions.

II. PARALLEL IMPLEMENTATION AND SOFTWARE TOOLS

A. Implementation Modes on Xeon-Phi

There are two important modes of implementation on a Xeon-Phi: offload and native. The offload mode executes the main function on a host CPU, and when it encounters the parallel computing zone, the parallel calculation is executed on the Xeon Phi. Its disadvantage resides in the lengthy offloading time between the host CPU and the Xeon-Phi that affects the program's performance [3]. With the native mode, the application runs entirely on the Xeon Phi. Based on previous

work of the authors [3], the native mode was found to be more efficient and is used in this work.

B. Open MultiProcessing (OPENMP)

In this paper, independent FFTs can be executed in parallel with no data dependencies between them. To parallelize this processing, we need to use the fork-join model of parallel computation, which places a tree of activities in parallel. The concurrent FFTs create sub-parallel tasks and synchronize them. To benefit from the fork-join model, OpenMP offers several functionalities. To begin a task, we need to use the directive "**pragma omp**". When this pragma is included, the code becomes a concurrent task, and is ready to be processed with the OpenMP thread [11]. OpenMP makes it possible to execute many threads in parallel on one or more CPUs and on the Xeon-Phi, join them together for HPC arrangement, and control the load of respective processors using synchronization directives. This model gets the number of the specified threads to execute the FFTs (Figure 1). During code execution, the directive "**pragma omp parallel**" generates other threads, each of which executing the code where the pragma is included. At the end of this pragma, the control returns to the master thread. OpenMP functions such as "**omp_get_thread_num**" and "**omp_get_max_threads**" are invoked as follows:

```
#pragma omp parallel num_threads(nThread)//Parallel Region
{int myID=omp_get_thread_num ();//Get the threads ID
status = DftiComputeForward(hand, &x[myID*len]);}//FFTs
```

To compile the code, we used the flag "**-openmp**". To highlight the number of threads to use in our application, we exported with the following statements the environment variable "**OMP_NUM_THREADS=100**", which is equal to the number of FFTs to execute:

```
icc -mmic -mkl -openmp fftmklnative.c -o native.MIC
micnativeloadex native.MIC -e "OMP_NUM_THREADS=100"
```

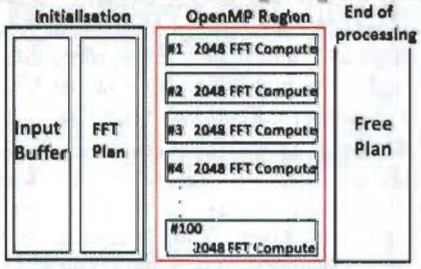


Figure 1 Concurrent FFTs with OpenMP

C. Message Passing Interface (MPI)

MPI is useful for message passing, network protocols and highly parallel programs. In fact, MPI messages can move rapidly across a TCP/IP network [11]. The executable that we run on many nodes needs to be delivered to these nodes by copying the executable on the Xeon/Xeon-Phi. This can be done manually, or by using a distributed file system within the Intel processors. In order to run an MPI program, we must use the command "**mpirun**", while indicating the hostname of the nodes and of the Xeon-Phi where we want to execute the concurrent FFTs. The network administrator provides an address (for example *aw-4r13-n09-mic0*), which targets Xeon Phi number 0 on the processor with the address *aw-4r13-n09*. We also need to mention the number of MPI processes for each host. The message in our case is one FFT executed on one process. The main communicator, which involves all processes, is the *MPI_COMM_WORLD*. MPI provides a group of functions that are very useful to develop MPI code. The function *MPI_Init_thread(&argc, &argv)* is set to initialize the parameters *argc* and *argv* of *MPI_Init_thread()*, which are the command line parameters of the C language. During execution, MPI removes all command parameters that the library can handle from *argv*, and reduces the number in *argc*. Therefore, we have to take care of these parameters after calling *MPI_Init()* to get the number of total processes and the current process ID invoked as follows:

```
MPI_Comm_rank (MPI_COMM_WORLD, &rank);/*get current process id */
MPI_Comm_size (MPI_COMM_WORLD, &size);/*get number of processes */
```

Unlike with OpenMP, MPI code is not totally synchronous. Thus, Xeon/Xeon-Phi clusters must obviously use MPI synchronisation. The processes are desynchronised from the beginning due to different start times and the fact that *MPI_Init()* may change the amount of time taken by each process. Another encountered cause of desynchronization is the OS noise, where the processes oddly share the CPU time with some MPI job. Therefore, to measure the true execution time of parallel FFTs and to mititigate OS induced jitter, a barrier should be placed as follows, before and after the FFTs block:

```
MPI_Barrier(MPI_COMM_WORLD);
mytime = MPI_Wtime();
{status = DftiComputeForward(hand,x);//FFTs compute
//printf( "FFT from processor %d" of %d, rank, size);}
mytime = MPI_Wtime() - mytime;
MPI_Barrier(MPI_COMM_WORLD);
```

The first *MPI_Barrier()* function can lead to poor synchronization of the various MPI processes. Indeed, it may happen that some of these processes appear at the next barrier early, while others appear late. in that case, early processes must wait for those that are late. There is no guarantee that all processes will leave the barrier simultaneously. The code contains a checkpoint when all the processes are inside the *MPI_Barrier()* function. Each process will call the same FFT function, with the same arguments, and therefore each process will have almost the same average processing time. One of the most useful communication directives we included is the "**reduction**". It is comparable to a reduction in shared memory, where an amount of data is dispatched from all processes to one process. We use *MPI_Reduce()* to get the sum of all the processes execution time and then divide it by the total number of processes to get the average time. We let the master process compute this mean time. This function called as shown below refers to operations such as summation, multiplication, minimum or maximum computation:

```
MPI_Reduce(&time, &Mean, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
  if (rank == 0) {Mean /= size;//Master Thread collects the data and
  printf("%lf\n",Mean);}//compute the Mean
MPI_Finalize();//End of MPI parallel region
```

It is not practical to get the execution time of each process, because the entire program runs in parallel and the individual process computation time will vary significantly.

After the OpenMPI, MKL and MIC libraries are loaded, it is time to export useful environment variables. The mixture of fabrics variable relies on the number of processes per node. If all processes begin on one node, the library uses shared memory (*shm*) communication. TCP/IP enables the network fabrics with *I_MPI_FABRICS=shm:tcp*. It is important to export *I_MPI_MIC=1* variables to enable the use of the Xeon-Phi coprocessor. To compile and run the MPI code on Xeon and Xeon Phi, we needed to follow some general instructions listed below:

```
mpiicc -mmic -mkl -o fftmpi.MIC fftmpi.c // Compile
```

scp fftmpi.MIC mic0:~/. // Copy the executable to MIC
mpirun -n 16 -host aw-4r13-n09 ./fftmpi //run on the Host Xeon
mpirun -n 100 -host aw-4r13-n09-mic0 -env LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH ./fftmpi.MIC //run on 1 MIC

This is comparable to compiling classic MPI C programs on a CPU by using the intel **mpiicc** directive. On a Xeon-Phi, the **"-mmic"** option must be inserted to work in the native mode. After compiling, the different libraries, executable files, and data must be transferred to the Xeon-Phi coprocessor by using the linux copy command **"scp"**. During this stage, the IP addresses of the CPU and the Xeon Phi should be allocated by exporting the environment variables as discussed. It is possible to execute MPI processes on both CPU and MIC in native mode, which is called heterogeneous programming. The different speeds of both CPU and MIC cores can create load balancing problems that must be found with memory leaks detecting tools such as *valgrind* and the segmentation faults detectors with debuggers. In our case, we have no major load balancing problem on the many cores of the Xeon Phi, because all the cores run at same speed in a symmetric mode. To use this symmetric mode on 2×MICs, the MPI executable should be copied to the 2×MICs and run there. In that case, load balancing consists of executing 50 FFTs on each MIC:

mpirun -n 50 -host aw-4r13-n09-mic0 -env LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH ./fftmpi.MIC :-n 50 -host aw-4r13-n09-mic1 -env LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH ./fftmpi.MIC //Run 2 MICS

The option *-host* represents the IP node to begin the MPI node. The option *-env* indicates the environment variable, and *./fftmpi* represents the path of the MPI program to be executed. *LD_LIBRARY_PATH $MIC_LD_LIBRARY_PATH* denotes the library path transfer from the host to the coprocessor. Because the path on a Xeon Phi is different from that on a CPU, the commas are used to divide various nodes. In the native mode each Xeon Phi is an independent node, which is helpful to prevent communications between the concurrent FFTs. As each Xeon Phi core can be considered a node, the MPI algorithm on each node can be divided into two categories: MPI code executing on Xeon Phi only and MPI code executing on heterogenous CPU/Xeon Phi together.

III. Hybrid MPI + OpenMP

In the hybrid mode, illustrated in Figure 2, we use MPI to go across the compute nodes and across the Xeon Phi within each node. However, we use OpenMP to execute tasks within the cores in each Xeon (CPU) or Xeon Phi. The advantage of the hybrid OpenMP+MPI method is that it can be used to speed up tasks for CPU clusters that already use MPI and OpenMP. Another benefit resides in avoiding offloads in the program by using only the native mode to avoid wasted communication times. The disadvantage is that MPI messaging has many end points, meaning that peer to peer communication between Xeon/Xeon Phi has to move through virtualized networks, which is perhaps less robust than physical communication between hosts. When writing a C/C++ program with, either MPI or OpenMP, certain basics must be considered including the storage location of the variables, the access method for the processes and threads and the way they scale across cores. All of these issues are magnified when using MPI and OpenMP together. We introduce OpenMP to MPI with the *MPI_THREAD_FUNNELED* directive to assure that the process is multi-threaded, so that there can be multiple OpenMP threads for every MPI process. The following directive ensures that only the master thread makes MPI calls for the OpenMP directive in order to avoid data races and deadlocks:

int required=MPI_THREAD_FUNNELED;
int provided; // Provided level of MPI threading support
MPI_Init_thread(&argc, &argv, required, &provided);

Instead of *MPI_Init()*, this time, the initialization goes with *MPI_Init_thread()* for our threaded code. In our experiments, we call an OpenMP pragma that builds a construct inside the MPI parallel zone executing 100 FFTs. Each FFT will be excuted by one MPI process and a number of OpenMP threads executing at the same time are invoked as follows:

#pragma omp parallel
{ ... //FFT execution on OpenMP Threads
printf("FFT from processor %d" of %d, thread %d\n", rank, size, omp_get_thread_num()); } MPI_Finalize();

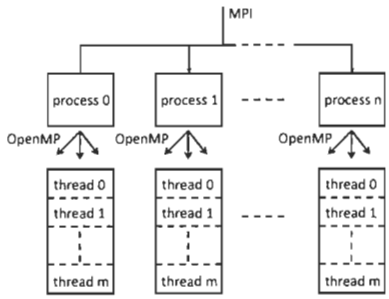


Figure 2 Hybrid mode on n processes and m OpenMP threads/process

IV. Evaluation of Performance and Results

A. Simulation environment and Platforms

The host platform is an Intel Xeon CPU E5-2650 operating at a 2.0 GHz frequency, with 64 GB DDR4 of RAM, 20 MB L3 cache memory, 8 cores and 16 threads. The operating system is CentOS 6.5 with the version 2.6.3 Kernel. The coprocessor platform, from *Calcul Quebec*, is a Xeon Phi coprocessor that contains 61 cores supporting 244 threads. Each core runs at a 1 GHz frequency and has an L2 cache memory size of 512 kB that interacts with cache memories of the other cores. The cores are interconnected by a high speed bidirectional ring. The coprocessor (Slave) connects with the Xeon processor (Master) using the PCI bus to exchange data [5]. We have two coprocessors per host that represent the central processing unit. The programming language is C/C++ and the main compiler used is the provided by Intel. The MKL MPI and OpenMP libraries are located in the parallel studio XE 2013. In each experiment, 2×10^3 size N FFTs calculated on complex float numbers were executed on Xeon Phi. To analyze the distribution of 2×10^3 computational times, we considered the mean, max and coefficient of variation as statistical metrics.

B. Results Discussion

The graphs in Figure 3 report the observed computation times for the seven implementation methods used in this work. The largest time was observed with the Hybrid OpenMP+MPI on 1×MIC with a mean time equals to 700 µs and a max of 800 µs. Using the heterogeneous implementation for the hybrid mode reduced the mean value to 31.3 µs. The Pure MPI results on MICs showed that the more we increase the number of MICs the better time we have, unfortunately, with the 3×MICs the max value is 93 µs which is greater than our target 66.7 µs derived from the LTE protocol, even if its average time of 54 µs is within the safe zone. The OpenMP multithreading method on

1xMIC appears to be in a good timing range, but it still does not satisfy the requirements. The best and lowest time is obtained on the pure MPI on the heterogeneous 3×MICs+CPU taking advantage of a good core load balancing. Figure 4 plots the mean, max, coefficient of variation (CV) values of the seven implementation methods. The pure MPI implementation on 3×MICs shows good scalability for the MICs distributed memory. The good performance is the result of optimized load balancing and synchronization between the nodes. When using the shared/distributed memory combination, the pure MPI on heterogeneous system appeared to be the most efficient. Network performance limits that of the pure MPI implementation. The OpenMP implementation also shows good scalability, but has a major disadvantage when compared to MPI. In fact, OpenMP parallelization needs a shared memory address space that restricts the scalability on the MIC. OpenMP works better on a cluster of CPUs. In heterogeneous MIC+CPU, the interaction between MPI and OpenMP is stronger than in MIC. MPI calls are made from parallel regions and threads where different MPI processes have to synchronize with each other. With Hybrid OpenMP+MPI on MIC, the OpenMP threads lacked the shared memory efficiency found on SMP platforms that led to a high mean, max, and coefficient of variation.

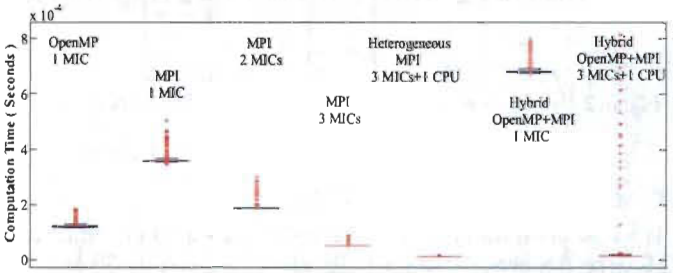


Figure 3 Boxplot of computation time for 100 concurrent FFTs (N=2048)

The CV plot demonstrates low values below 0.01 for all the FFT sizes for the MPI on Heterogeneous 3×MICs+CPU, unlike the hybrid OpenMP+MPI results that exceeded 0.5. The CV values with MPI on 1, 2 and 3 MICs and OpenMP remain close to 0.1. Hybrid OpenMP+MPI creates a domain decomposition as a two level application. In fact, hybridizing OpenMP+MPI introduces hard challenges due to thread placement and ordering problems, whereas pure MPI adapts easily with the platform's topology.

V. Conclusion

In this paper, we explored various means of implementing a workload composed of 100 FFTs using OpenMP on 1xMIC, MPI on one, two and three MICs, MPI on heterogeneous 3×MICs+CPU, Hybrid OpenMP+MPI on 1×MIC and 3×MICs+CPU. The solution based on pure MPI executing on an Heterogeneous Xeon/Xeon Phi cluster respected the LTE constraint of 66.7 µs, and made full use of the system architecture. It is better than a pure MPI implementation on the MICs cluster, OpenMP and Hybrid OpenMP+MPI. This paper contributes to show how LTE can be effectively implemented on heterogeneous clusters for future cloud radio access network.

Acknowledgments

The authors wish to express their thanks to the ReSMiQ (*Regroupement Stratégique pour la Microélectronique du Québec*) and the Natural Sciences and Engineering Research Council of Canada for their financial support, *Calcul Quebec* for providing access to HPC equipment and their dedicated help, specifically Dr. Bart Oldeman from McGill HPC team.

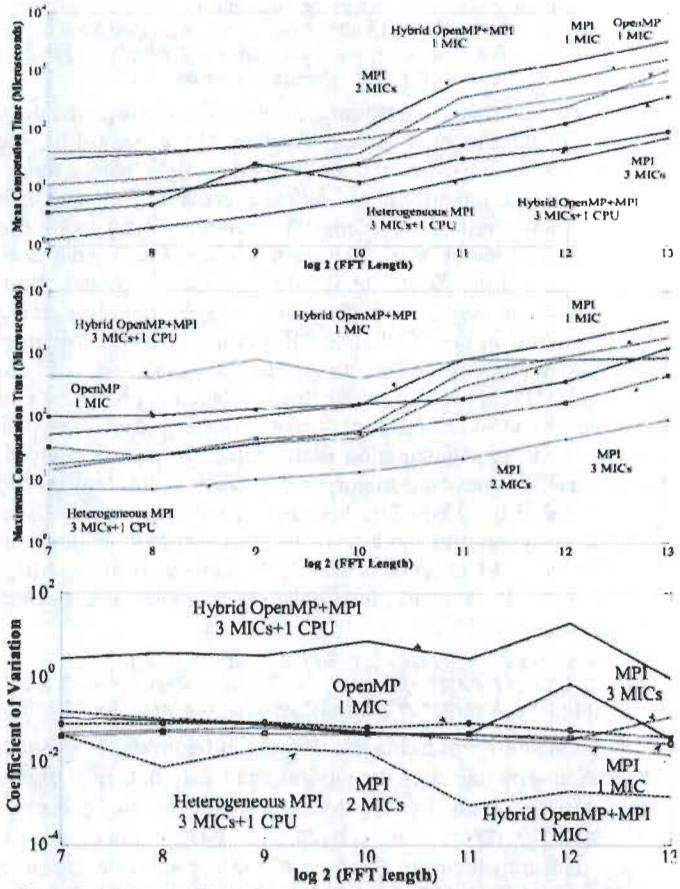


Figure 4 Comparative plots of time required by seven considered implementation modes on Xeon Phi for 100 concurrent FFTs: a) mean computational time b) maximum computation time, and c) coefficient of variation

References

[1] Y. Soo Cho "MIMO-OFDM Wireless Communications With Matlab," John Wiley& Sons, 2010 (chapters 4 and 5)

[2] R. Mego, T. Fryza, "*Performance of parallel algorithms using OpenMP.*" Conf. Radioelektronika, Czech Republic, April 2013, pp. 236-239

[3] M. Khelifi, D. Massicotte, Y. Savaria, "*Parallel Independent FFTs Implementation on Intel Processors and Xeon Phi for OFDM and LTE systems.*" Conf, Nordic Circuits and Systems, Oslo, Oct. 2015, pp. 1-4

[4] H. Kamal, A. Wagner "*FG-MPI: Fine-grain MPI for multicore and clusters,*" Int. Symp. on Parallel & Distributed Processing, Workshops and Phd Forum, Atlanta, April 2010, pp.1-8

[5] H. Zhou, X. Wang, "*A Parallel Computing Component for Linux Cluster with Threads Binding Supports,*" Computational Intelligence and Software Engineering, Wuhan, Dec. 2010, pp. 1-4

[6] G.Locharla, S.Kumar, K Mahapatra, S.Ari,"*Implementation of Input Data Buffering and Scheduling Methodology for 8 parallel MDC FFT,*" Int. Symp. on VLSI Design and Test, India, June 2015, pp.1-6

[7] Y.Li, W.Shen, A.Shi, L.He, D.Zhao "*MPI and OpenMP Paradigms on Cluster with multicores and its application on FFT,*" Int. Conf. on Computer Design and Application, Qinhuangdao, June 2010, pp.V1-23 – V1-26

[8] M.D.Jones, R.Yao "*Parallel programming for OSEM reconstruction with MPI, OpenMP, and hybrid MPI-OpenMP.*" Nuclear. Symp. Conf. Record, Rome, Oct. 2004, pp.3036-3042

[9] P.D'Ambra, M.Danelutto, D.di Serafino, M.Lapegna, "*Integrating MPI-based numerical software into an advanced parallel computing environment,*" Conf. on Parallel, Distributed and Network-based Processing, Italy , Feb. 2003, pp.283-291

[10] K. G.Lenzi, F.A.P. Figueiredo, J.A.B. Filho, "*Fully Optimized Code Block Segmentation Algorithm for LTE-Advanced,*" Wireless Comm. and Networking Conf., Shanghai, 2013, pp. 3312-3317

[11] E.Wang, Q.Zhang, G. Zhang, X.Lu, Q.Wu, "High Performance Computing on Intel Xeon Phi," Springer Inc., 2012 (chapters 1 to 6 and 11)