Université Sidi Mohamed Ben Abdellah
Faculté des Sciences et Techniques Fès
Département Génie Electrique

Mémoire de Projet de fin d'étude

Préparé par

Dounia OUDRHIRI HASSANI

Pour l'obtention du diplôme
Ingénieur d'Etat en
SYSTEMES ELECTRONIQUES & TELECOMMUNICATIONS

Intitulé

Etude et évaluation des performances d'OpenMP sur des processeurs multi-cœur Application à la chaine RADAR

Encadré par :

Mr Philippe LEGALL (THALES)
Mr Mounir MAMDOUHE (MASCIR)
Pr Ali AHAITOUF
Pr Fatima ERRAHIMI

Soutenu le Mercredi 29 Juin 2011, devant le jury composé de :

Pr Ali AHAITOUF	**Encadrant**
Pr Fatima ERRAHIMI	**Encadrant**
Mr Philippe LEGALL	**Encadrant (THALES)**
Mr Mounir MAMDOUHE	**Encadrant (MASCIR)**
Pr Anass MANSOURI	**Examinateur**
Pr Hicham GHENNIOUI	**Examinateur**

ANNEE UNIVERSITAIRE 2010-2011

Université Sidi Mohamed Ben Abdellah
Faculté des Sciences et Techniques Fès
Département Génie Electrique

Notice Bibliographique

Auteur	Dounia Oudrhiri Hassani
Année	2011
Etablissement	FST de Fès
Spécialité	Ingénieur d'Etat en Systèmes Electroniques et Télécommunications
Entreprises	MASCIR THALES AIR SYSTEMS
Cadre du rapport	Rapport du stage de fin d'études
Titre du projet de stage	**Etude et évaluation des performances d'OpenMP sur des processeurs multi-coeur Application à la chaine RADAR**
Encadrement entreprise	M. Philippe LEGALL:Manager Equipe THALES AIR SYSTEMS M. Mounir MAMDOUHE : Ingénieur systèmes embarqués MASCIR
Suivi pédagogique	M. Ali AHAITOUF : Professeur de l'enseignement supérieur FST de Fès Mme Fatima ERRAHIMI : Professeur de l'enseignement supérieur FST de Fès
Date de début	07 Février 2011
Date de fin	07 Juillet 2011

ANNEE UNIVERSITAIRE 2010-2011

Université Sidi Mohamed Ben Abdellah
Faculté des Sciences et Techniques Fès
Département Génie Electrique

Dédicaces

Je dédie ce mémoire au bon Dieu
A mes chers parents
A mon frère et ma sœur
A toute ma famille
A mes professeurs
A mes amis

Remerciements

ANNEE UNIVERSITAIRE 2010-2011

Avant d'entamer la réalisation de ce rapport, je tiens à signaler que ce travail n'a pu aboutir sans l'aide de Dieu et le soutien de plusieurs personnes qui ont contribué pleinement à la réussite de ce travail.

Il faut noter tout d'abord que ma formation à la FST de Fès a été un de mes plus forts appuis. Aussi, j'espère avoir réussi à être à la hauteur de l'attente de mes responsables et avoir su gérer au mieux les différentes tâches dont on m'a donné la charge.

Je tiens à remercier Mr Mouhcine ZOUAK, Mr M'hammed LAHBABI, Mr Ali AHAITOUF, Mme Fatima ERRAHIMI, tous nos Professeurs de la FST, et le personnel de la FST.

Mr Philippe LEGALL, mon encadrant de THALES, Mr Mounir MAMDOUHE,Mr Abdessamad KLILOUet Mr François BOURZEIX, l'équipe encadrante au MASCIR, pour leurs aide précieuse, leurs conseils prodigieux, leur confiance et leur soutien morale. Tous les membres du département systèmes embarqués et également tous le personnel de MASCIR pour leur aide et leur accueil chaleureux. Veuillez accepter l'expression de ma grande reconnaissance, ma gratitude et mon profond respect.

Ensuite, je tiens à exprimer mes remerciements aux membres du jury pour avoir accepté d'évaluer mon travail. J'en suis très honorée.

Je n'oublierai sans doute pas de remercier mes parents qui m'épaulent dans toutes mes démarches et qui étaient toujours là pour me soutenir dans les moments difficiles comme dans les moments de joie.

Que tous ceux qui ont contribué, de près ou de loin, à la réalisation du présent travail, trouvent ici l'expression de mes meilleurs sentiments.

Table des matières

Introduction générale

De nos jours, l'industrie de la défense se tourne vers des radars à usages multiples qui doivent répondre aux besoins du contrôle aérien, la sécurité intérieure, et l'observation météorologique. Ces radars sont l'avenir de l'industrie et seront entièrement numériques.

Puisque le comportement de ces radars sera entièrement contrôlé par logiciel, on aura besoin d'exécuter des algorithmes de traitement de signal très lourds. Pour répondre à ces conditions, l'utilisation d'une machine parallèle s'avère indispensable. Donc il faut adapter ces algorithmes pour fonctionner de façon parallèle.

Dans ce cadre, THALES a lancé une thèse en collaboration avec l'ENSA de Marrakech et la fondation MAScIR, visant à concevoir des modèles parallèles pour les algorithmes de traitement de signal qui vont être utilisés sur les radars. Ainsi le sujet de mon stage vient en support pour la thèse, pour aider à montrer comment certains modèles pourront être réalisés avec l'interface de programmation OpenMP.

L'objectif du stage est dans un premier temps, l'étude de l'API OpenMP et des différentes fonctionnalités qu'il propose en utilisant des cas d'utilisation « use case »proposés par THALES et dans un deuxième temps, appliquer OpenMP aux algorithmes de la chaine radar en proposant plusieurs implémentations de parallélisme.

Vue la nature des traitements radar, Le microprocesseur le plus adapté pour construire la machine parallèle est le DSP. Particulièrement, les DSPs de Texas Instruments (TI) sont les plus utilisés sur le marché, car ils proposent des performances très importantes ainsi qu'un support software robuste qui baisse drastiquement le temps de développement logiciel.

Malheureusement et à cause d'un imprévu, la bibliothèque d'exécution d'OpenMP s'est avérée indisponible sur DSP Texas pendant la période du stage. Pour contourner ce problème THALES a choisi de faire l'étude sur le microprocesseur Core2Duo d'Intel.

Chapitre 1:Contexte du stage

De nos jours le parallélisme est devenu indispensable à cause des traitements et des calculs importants. Le parallélisme consiste à implémenter des architectures d'électronique numérique et des algorithmes spécialisés pour celles-ci, permettant de traiter des informations de manière simultanées. Le but de ces techniques est d'effectuer le plus grand nombres d'opérations dans le plus court temps possible.

Par ailleurs, les architectures parallèles sont devenues le paradigme dominant pour tous les ordinateurs depuis les années 2000. En effet, la vitesse de traitement qui est liée à l'augmentation de la fréquence des processeurs connait des limites en raison d'une augmentation de la production thermique qui provoque des erreurs de calculs.

Dans ce cadre, THALES AIR SYSTEMS souhaite étudier le nouveau standard OpenMP (Open Multi-Processing) qui est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette étude a pour objectif principale l'intégration de cet outil dans la chaine de traitement des futurs générations de radars.

Dans ce chapitre, on présentera les deux entreprises avec lesquelles on a réalisé le projet, à savoir Mascir et Thales. Ainsi qu'une description détaillée du projet et du cahier des charges.

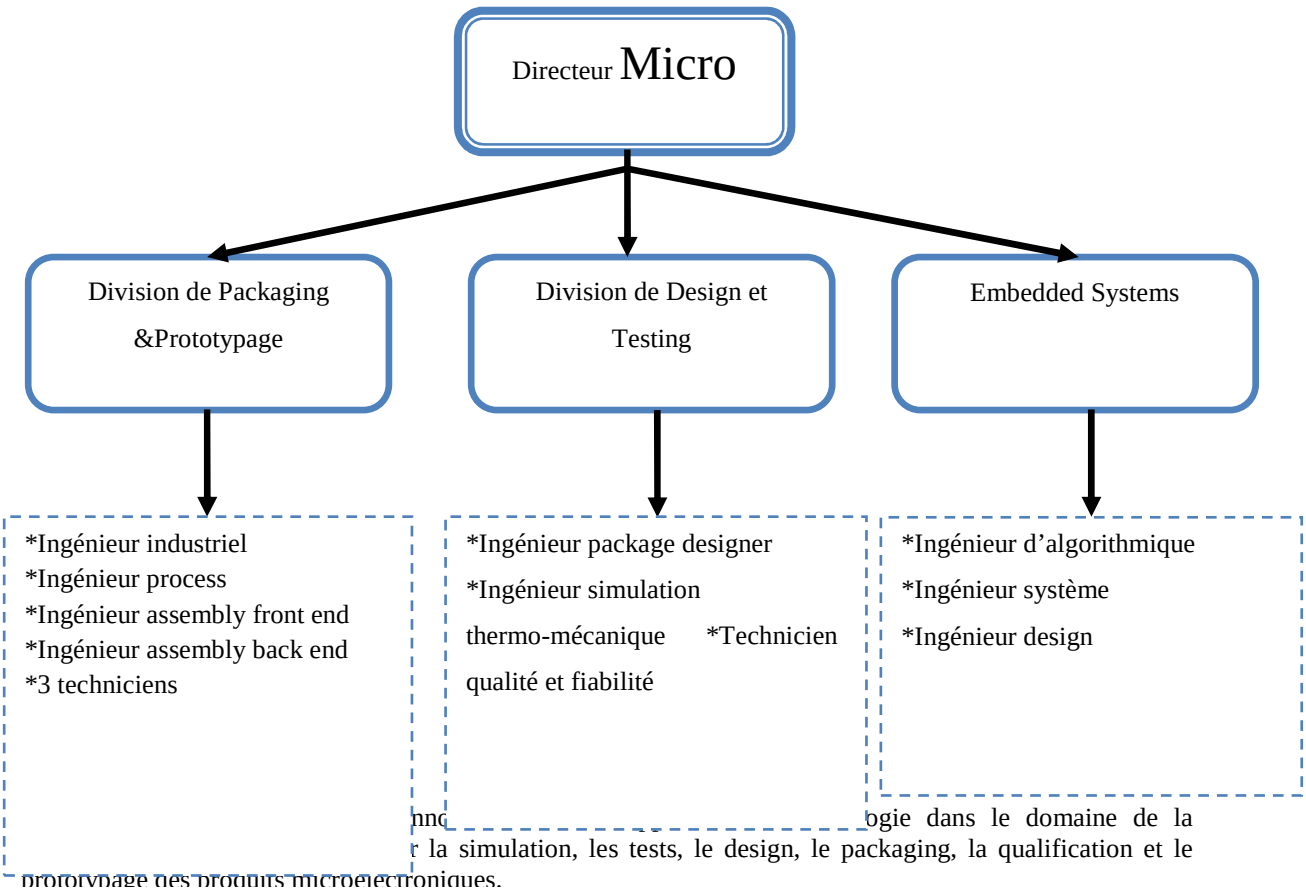
Présentation de Mascir

MAScIR (Moroccan foundation for Advanced Science Innovation and Research) est une fondation à utilité publique créée en 2007 par le gouvernement marocain pour promouvoir la recherche et développement. Son local est sur Rabat Shore Center, Technopolis.
La nanotechnologie, la biomédecine, et lamicroélectronique représentent les institutions de recherche de cette fondation :

- MAScIR Micro : créée en 2008 pour la recherche dans le domaine de la microélectronique
- MAScIR bio : travaille dans la biotechnologie des drogues et biocides
- INanoTech : forme le troisième membre de MAScIR dans le domaine de la nanotechnologie

MASCIR micro

La figure suivante montre la hiérarchie du département Micro de la fondation MAScIR:



nnc ogie dans le domaine de la
r la simulation, les tests, le design, le packaging, la qualification et le
prototypage des produits microélectroniques.

Figure 1: Hiérarchie du département Micro

business dans les domaines suivants :

- L'intégration et la miniaturisation des systèmes microélectroniques

- L'analyse de fiabilité et défaillance des produits
- Modélisation des systèmes complexes
- Prototypage et industrialisation des produits innovants
- Industrialisation des idées et résultats académiques

MAScIR possède plusieurs laboratoires équipés de technologie avancée :

- Chambre blanche
- Laboratoire optique
- Laboratoire électronique

Présentation de Thales

Pour répondre à la forte progression de la demande de sécurité et saisir les opportunités de croissance sur ses marchés, Thales mobilise des savoir-faire centrés sur les systèmes d'information critiques, les moyens de communication sécurisés, les systèmes de supervision et les équipements de détection. Le Groupe propose une gamme complète de solutions et de technologies permettant d'apporter des réponses adaptées aux besoins de ses clients gouvernementaux et institutionnels, avec lesquels il noue des relations de long terme, fondées sur la proximité et la confiance indispensables à la conduite de projets complexes dans le domaine de la sécurité : avec les forces de sécurité militaires et civiles en premier lieu, mais aussi avec les autres autorités publiques, ainsi que les grands opérateurs d'infrastructures sensibles et les grands avionneurs civils.

L'organisation du Groupe Thales

Présent dans les domaines civils et militaires, Thales est organisé en trois grands domaines d'activité définis par leurs marchés : l'Aéronautique et l'Espace, la Défense et la Sécurité. Un ensemble cohérent irrigué par une expertise technologique commune de très haut niveau et des savoir-faire transversaux au service de ses clients.

Figure 2: Domaines d'activité de Thales

Chiffres clés

Le tableau ci-dessous donne les principales informations à propos de THALES

Création	**1892**
Siège social	Neuilly-sur-Seine, France
Direction	Dennis Ranque, PDG
Activités	Aéronautique, Défense, Technologies de l'information
Effectif	67 028 effectifs gérés au 31/12/07
Capitalisation	7.5 Mds € (Septembre 2008)
Chiffre d'affaires	12.3 Mds €
Résultat net	887 M €

Tableau 1: Informations à propos de THALES

Jusqu'au 31 mai 2009, la répartition de l'actionnariat du groupe Thales est donnée par le graphe suivant:

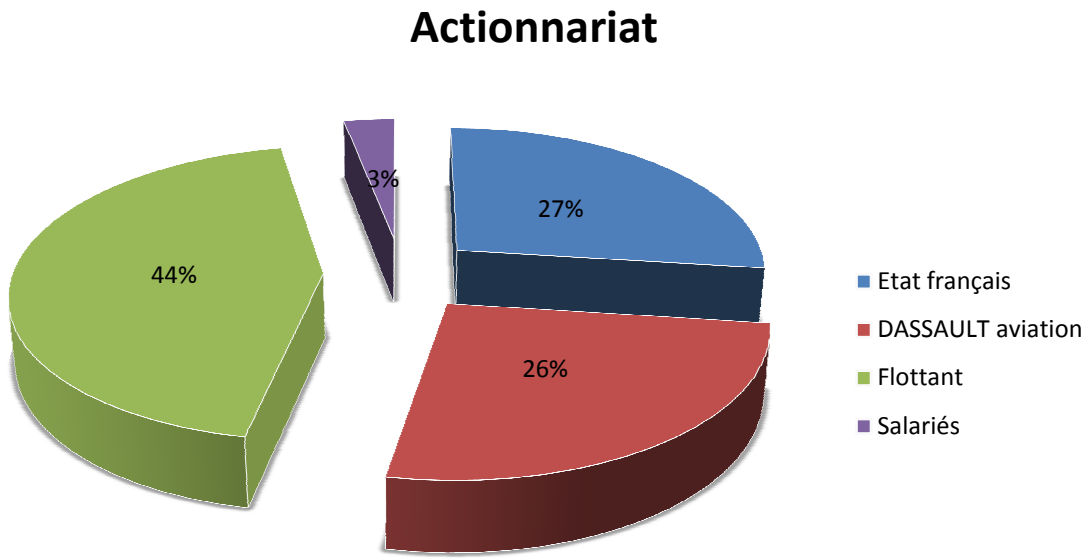


Figure 3: Répartition du capital de Thales au 31 mai 2009

Présentation de THALES Air Systems

La division Air Systems du Groupe Thales conçoit et fournit des solutions globales de sécurité aérienne dans les domaines civil et militaire. Dans le domaine de l'aviation civile, la division développe des systèmes de gestion du trafic aérien, des outils d'aide à l'atterrissage et à la navigation ainsi que des solutions pour la sécurité des zones aéroportuaires. Dans le cadre de ses activités de défense aérienne, la division propose une gamme complète de radars de surface, des systèmes de conduite des opérations aériennes, des solutions de protection du champ de bataille et des sites sensibles ainsi que des systèmes d'armes antiaériens, terrestres et navals. En appui de cette offre, la division propose une gamme complète de services de maintenance, de rénovation et d'extension de durée de vie ainsi que des services d'ingénierie logistique permettant de concevoir dès la phase de développement, la solution de maintenance adaptée au système vendu.

Thales Air Systems est :

- Leader mondial des systèmes de gestion du trafic aérien (hors USA)
- Leader mondial des centres de commandement et de contrôle des opérations aériennes
- N° 3 des radars de surveillance naval et terrestre
- Premier fournisseur européen de fusées, d'autodirecteurs de missiles et de munitions guidées

Présentation du projet

Objectifs du projet

L'objectif principal de mon stage est de supporter la thèse doctorale, dans le sens de l'étude du nouveau standard OpenMP. L'étude consiste en la compréhension du fonctionnement de ce dernier dans un premier lieu, ensuite décrire les composants qui pourront apporter des performances dans une architecture parallèle en accélérant au maximum le temps d'exécution. Et enfin implémenter certains modèles parallèles sur les blocs de la chaine radar qui seront utilisés par la suite dans la thèse.

Pour récapituler, les objectifs du stage peuvent être résumés comme suit :

1. Etude d'OpenMP
2. Etude des fonctionnalités d'OpenMP en sélectionnant les plus performantes.
3. Application aux algorithmes du traitement cohérent de la chaine radar, à savoir :
   - La formation de faisceaux par le calcul
   - La compression d'impulsions
   - Le filtrage Doppler

Déroulement et répartition des tâches

Pendant la période du stage on a été amené à effectuer plusieurs tâches dont principalement:

Tâche1: Réalisation des différentes actions qui contribuent au bon déroulement du stage

- Dépôt de programmes et livrables sur une zone de partage sous le serveur de MAScIR.
- Rédaction de rapports d'avancement hebdomadaire « weekly report » décrivant les points abordés au cours de la semaine passée, et ceux qui vont être abordés par la semaine qui suit.
- Présentations quasi mensuelles de l'état d'avancement du projet.
- Rédaction de documents décrivant les résultats du projet.

Tâche 2: Etude de OpenMP

- Etude des modules proposés par OpenMP et les fonctionnalités de chacun
- Etude des directives de compilation
- Etude des routines qui font appel à la bibliothèque d'exécution
- Etude des variables d'environnement
- Illustration par des programmes exemple en utilisant des use case proposés par THALES

Tâche 3: Compilation OpenMP

- Etude de la chaine de compilation avec OpenMP
- Etude de la structure du compilateur en général
- Etude du compilateur OpenUH, un compilateur open source de OpenMP

Tâche 4: Application à la chaine radar

- Etude de la chaine radar, particulièrement la partie traitement cohérent qui comporte trois blocs : La formation de faisceaux par le calcul, la compression d'impulsions et le filtrage Doppler
- Implémentation des algorithmes des blocs radar avec des modèles en Matlab et en langage C
- Implémentation d'OpenMP sur ces algorithmes en implémentant plusieurs modèles de parallélisme

Pour mettre au point ce projet, nous avons été placés dans les locaux du MAScir au technopole de Rabat.

Un PC doté d'un processeur Intel Core2Duo E8400, dont les principaux caractéristiques sont décrits dans le tableau 2,a été mis en notre disposition dans lequel on a installé deux partitions. La première contenait le système d'exploitation Windows 7 et Visual Studio 2010 dans lequel on a intégréle compilateur d'Intel (Intel C++ v12.0) qui supporte la dernière versiond'OpenMP 3.0. La deuxième partition contenait le système d'exploitation Linux distribution Ubuntu avec le compilateur GCC. Cette dernière partition a été mise en place juste pour la validation des tests avec la version Microsoft.

Concernant le choix de l'environnement de développement, on a choisi d'écrire le code de la chaine radar dans Visual Studio 2010. Ce choix a été fait parce que d'une part c'est un IDE souple et facile à utiliser et d'autre part il organise les fichiers dans un concept de projet.

Processeur	E8400
Nombre de cores	2
Fréquence d'horloge	3GHZ
L2 cache	6MB
Jeu d'instructions	64 bits
Performances	24 GFLOPS
Technologie	45 nm

Tableau 2: Principales caractéristiques du processeur Intel Core2Duo E8400

Planning du projet

La planification du projet doit bien englober et cerner toutes les étapes d'étude et réalisation. Vu que le projet est réparti sur plusieurs axes, on a bien été poussé à adopter une planification temporelle pour bien cerner les étapes de réalisation, ainsi que les livrables à produire. Cette planification est conforme aux phases du projet ainsi que les objectifs attendus, au cours de cette période, un ensemble de points de contrôle est mis en place

pour bien établir un suivi du déroulement des étapes de projet, ainsi que la discussion et validation des livrables. La durée du stage est de 5 mois à partir du 7 Février 2011.

La figure suivante illustre cette planification:

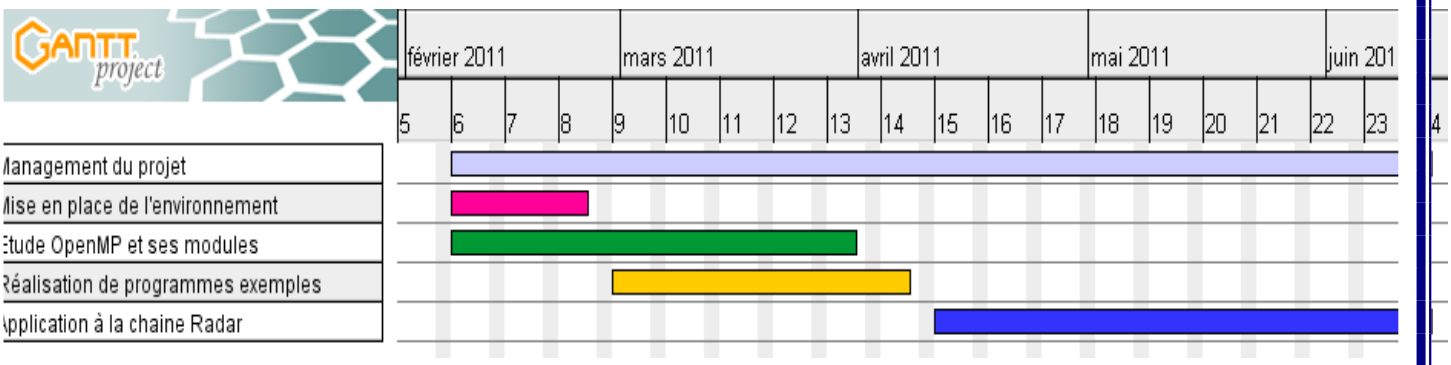


Figure 4: Planning du projet

Chapitre 2: Architectures parallèles

Durant des décennies, la plupart des ordinateurs étaient séquentiels (voir figure 5), le processeur exécutait les instructions une par une telle qu'elles lui étaient fournies. Le logiciel aussi était écrit pour une série de calcul, pour être exécuté sur un seul ordinateur ayant une seule unité centrale de traitement (CPU).

La loi de Moore a une limite si on reste sur des machines à un seul processeur et sur une algorithmique séquentielle. Aussi, la vitesse de traitement qui est liée à l'augmentation de la fréquence des processeurs connait des limites en raison d'une augmentation de la production thermique qui provoque des erreurs de calculs. Alors, il va falloir trouver un autre moyen pour continuer à accroître la puissance de calcul.

D'où le parallélisme qui consiste à implémenter des architectures d'électroniques numériques et les algorithmes spécialisés pour celles-ci, permettant de traiter des informations de manière simultanées.Le but de ces techniques est d'effectuer, par une machine, le plus grand nombre d'opérations dans le plus petit temps possible. Pour ce faire, les opérations doivent être faites en parallèle, c'est-à-dire simultanément au sein de plusieurs unités de traitement (voir figure 6). La tâche à effectuer est décomposée en de multiples sous-tâches qui sont exécutées en même temps et qui composent chacune des architectures parallèles [2].

L'idée est de faire coopérer plusieurs processeurs pour réaliser un calcul.

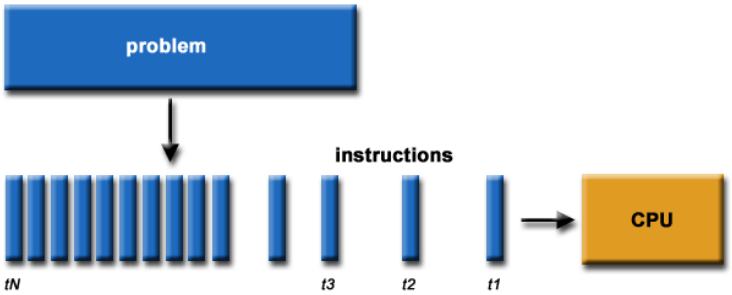


Figure 5: Représentation du calcul séquentiel

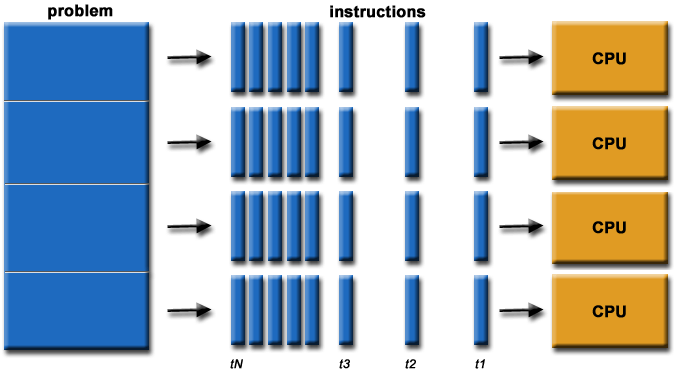


Figure 6: Représentation du calcul parallèle

Le principal avantage du parallélisme est la rapidité.En théorie, pour N processeurs, le temps de calcul est divisé par N. Tandis que les difficultés résident principalement dans la gestion du partage des tâches et l'échange d'information (tâches non indépendantes).

Classification des architectures parallèles

Une bonne connaissance de la classification des architectures parallèles permet une bonne approche dans la recherche des solutions aux problèmes de calcul intensif.

La Taxinomie de Flynn, proposée par l'américain Michael J. Flynn est l'un des premiers systèmes de classification des ordinateurs créés. Il évalue tant le matériel que le logiciel. Les programmes et les architectures sont classés selon le type d'organisation du flux de données et du flux d'instructions [1].

SISD*Single Instruction, Single Data*
Il s'agit d'un ordinateur séquentiel qui n'exploite aucun parallélisme, tant au niveau des instructions qu'à celui de la mémoire. Cette catégorie correspond à l'architecture de von Neumann. Elle supporte le parallélisme apparent.

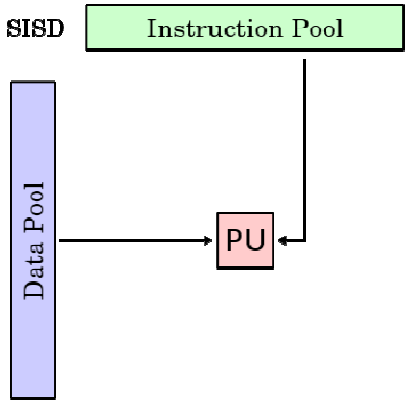


Figure 7: Représentation SISD

SIMD*Single Instruction, Multiple*
Il s'agit d'un ordinateur qui utilise le parallélisme au niveau de la mémoire.

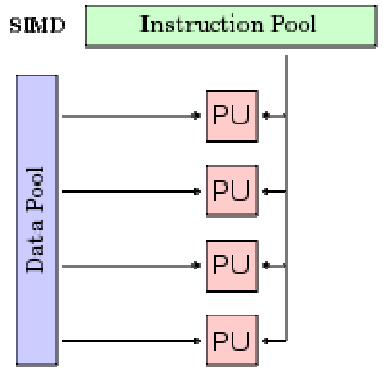


Figure 8: Représentation SIMD

MISD*Multiple instructions, Single Data*
Il s'agit d'un ordinateur dans lequel une même donnée est traitée par plusieurs processeurs en parallèle. Il existe peu d'implémentations en pratique. Cette catégorie peut être utilisée dans le filtrage numérique et la vérification de redondance dans les systèmes critiques.

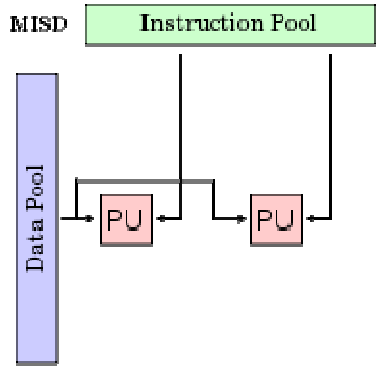


Figure 9: Représentation MISD

MIMD*Multiple instructions, Mu*
Dans ce cas, plusieurs processeurs traitent des données différentes, car chacun d'eux possède une mémoire distincte. Il s'agit de l'architecture parallèle la plus utilisée.
Nous rencontrons principalement les deux variantes suivantes.

- **MIMD à mémoire partagée**

Les processeurs ont accès à la mémoire comme un espace d'adressage global. Tout changement dans une case mémoire est vu par les autres CPU. La communication inter-CPU est effectuée via la mémoire globale. Tel est le cas pour notre sujet sur OpenMP qui se situe dans cette catégorie.

- **MIMD à mémoire distribuée**

Chaque CPU a sa propre mémoire et son propre système d'exploitation. Ce second cas de figure nécessite un middleware pour la synchronisation et la communication.

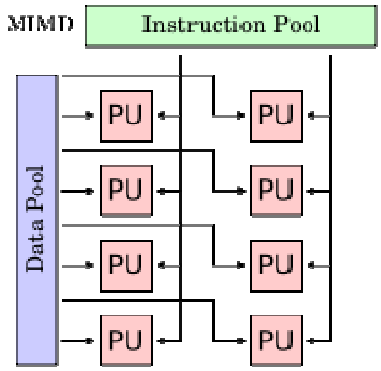


Figure 10: Représentation MIMD

Organisation de la mémoire

Mémoire distribuée

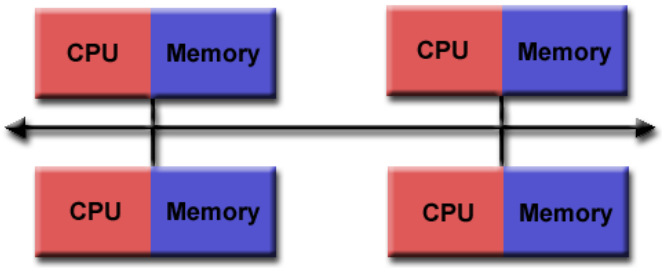


Figure 11 : Mémoire distribuée

Une architecture à mémoire distribuée se présente comme un espace mémoire associé à chaque processeur. L'accès à la mémoire du processeur voisin se fait par échange de messages à travers le réseau entre les processeurs. Alors, les algorithmes utilisés devront minimiser les communications.
Parmi les avantages de cette architecture, la scalabilité de la mémoire avec le nombre de processeurs, la rapidité d'accès que possède chaque processeur à sa propre mémoire et enfin lecoût réduit et lafacilité de construction.
Tandis que les difficultés résident dans la gestion des communications qui doit être assurée par le programmeur. Ce qui amène à un risque d'erreurs élevé et une complexité des algorithmes[2].

Mémoire partagée

Une architecture à mémoire partagée se présente comme un espace mémoire global visible par tous les processeurs. Les processeurs auront leur propre mémoire locale(cache,...) dans laquelle sera copié une partie de la mémoire globale. La cohérence entre ces mémoires locales devra être gérée par le hardware, le software et parfois l'utilisateur.

Il existe deux classes d'architectures parallèles à mémoire partagée basées sur le temps d'accès à la mémoire[2].

Accès uniforme à la mémoire (UMA)

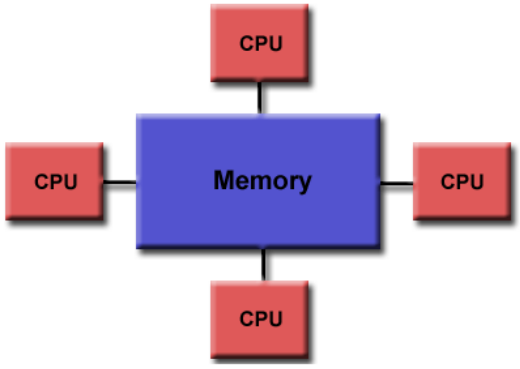


Figure 12: Mémoire partagée UMA

Ce type de mémoire partagée est connu sous le nom de machines SMP (Symmetric Multi processors). Tous les processeurs sont identiques ainsi que le temps d'accès à la mémoire depuis chaque processeur. Certains systèmes sont CC-UMA (cache coherent UMA), si un processeur met à jour une variable dans la mémoire globale, tous les processeurs connaissent cette mise à jour.

Accès non uniforme à la mémoire (NUMA):

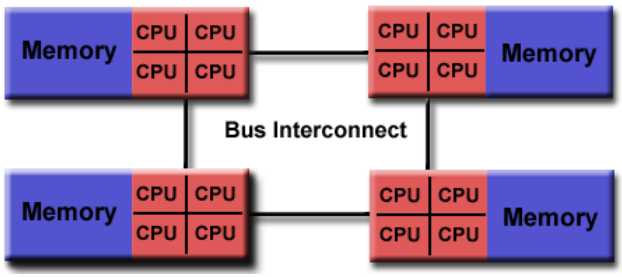


Figure 13: Mémoire partagée NUMA

Cette architecture est construite à partir d'un lien physique entre deux ou plusieurs machines SMP. Un noeud SMP accède directement à la mémoire d'un autre noeud SMP sans échange de message. Le temps d'accès à la mémoire globale n'est pas uniforme,l'accès à la mémoire via le lien physique est plus long. Cette machine peut avoir un système de cohérence des caches: CC-NUMA.

Multithreading et Multi-process

Un processus est une opération complexe exécutable par un ordinateur et défini par un ensemble d'instructions à exécuter (un programme), un espace d'adressage en mémoire vive et éventuellement d'autres ressources.
Un processus possède au moins un thread (qui exécute le programme principal, habituellement la fonction main()). Il peut aussi posséder plusieurs threads[3].

Tandis qu'un thread est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire. Par contre, tous les threads possèdent leur propre pile d'appel.
Les ressources allouées à un processus (temps processeur, mémoire) sont partagées entre les threads qui le composent[4].

Une des premières accélérations possibles a été le multithreading. En apparence le programme donnait l'impression de simultanéité car il permettait de lancer plusieurs actions en même temps, mais en termes de durée d'exécution totale, celui-ci était plus lent car les tâches s'alternaient au gré de l'ordonnanceur. La popularisation des machines multi-cœurs permet enfin de concevoir des applications réellement parallèles.

A l'heure actuelle, très peu de logiciels utilisent entièrement les avantages de cette technologie ce qui engendre même une baisse des performances. Ce constat vient du fait que le développement des applications parallèles nécessite de refondre des parties entières du code, savoir distinguer quelles parties sont parallélisables et lesquelles ne le sont pas. Toutefois peu à peu, les développeurs ainsi que les entreprises s'intéressent à cette nouvelle technologie notamment pour les programmes nécessitant de grandes puissances de calcul.

Lafigure suivante présente les différences entre le calcul séquentiel et le multithreading sur une machine ne possédant qu'un cœur.

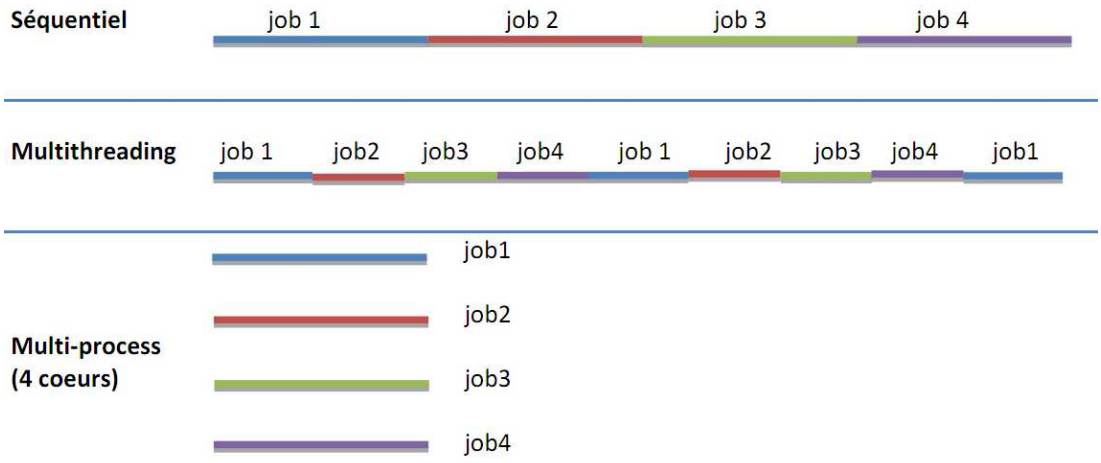


Figure 14 : Différence entre le multithreading et le multi-process

On voit sur la figure ci-dessus la différence en termes de temps entre le programme séquentiel et le multithreading est nulle, par contre en comparant avec le multi-process, on a bien divisé par quatre, puisque chaque cœur se voit, attribuer sa propre tâche[5].

Conclusion
OpenMP peut faire du multithreading et du multiprocess selon si le programme est exécuté respectivement sur un seul processeur où plusieurs processeurs. Il se situe dans l'architecture de type MIMD à mémoire

partagée plus précisément dans la classe UMA où les différents processeurs ont un temps d'accès uniforme à la mémoire.

Chapitre 3: Présentation d'OpenMP

OpenMPest un APIqui a été développé pour permettre une programmationparallèle portable sur une architecture à mémoirepartagée (SMP). Cette API est supportée sur de nombreuses plateformes, incluant

Unix et Windows NT, pour les langages de programmation C/C++ et Fortran.Définit conjointement par un groupe de matériel informatique et de grands éditeurs de logiciels nommé ARB, OpenMP donne aux programmeurs une interface simple et flexible pour développer des applications parallèles pour les plates-formes allant de l'ordinateur de bureau au superordinateur.

L'API estconçue pour permettreune approche progressive de parallélisationd'un code existant. Dans lequel des partiesdu programmesont parallélisés, éventuellement par étapes successives.OpenMPa étéégalementconçupour permettre auxprogrammeursd'avoir unseul ensemble defichiers sourcecontenant la version séquentielle du codeainsi que la version parallèle[6].

Les modules d'OpenMP peuvent être classés en 3 catégories :

- Directives (pragma) de compilation
- Fonctions (routines) de bibliothèque
- Variables d'environnement

Architecture d'OpenMP

Modèle d'exécution

A l'exécution du programme, le système d'exploitation construit une région parallèle suivant le modèle « fork and join » schématisé dans la figure 15.
A l'entrée d'une région parallèle, lethread maitre crée/active (fork) des threads esclaves qui disparaissent/s'assoupissent en fin de région parallèle (join) pendant que le thread maitre poursuit seule l'exécution du programme jusqu'à l'entrée de la région parallèle suivante.

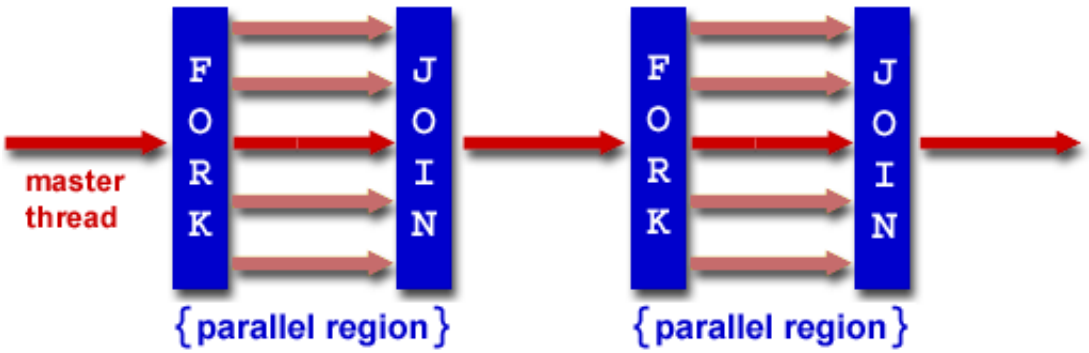


Figure 15: Modèle d'exécution d'OpenMP

À la fin de chaque région parallèle, les threads d'une même équipe s'attendent au moyen d'une barrière (implicite). Chaque thread à un identifiant unique, l'identifiant du thread maître est toujours égale à 0.

Les threads peuvent avoir leur propre « vue de la mémoire », cacher leurs données et ne sont pas tenus de maintenir une cohérence avec la mémoire réelle. Quand il ne faut pas que tous les threads partagent une variable, c'est au programmeur de s'assurer que la variable est privée pour chaque thread[7].

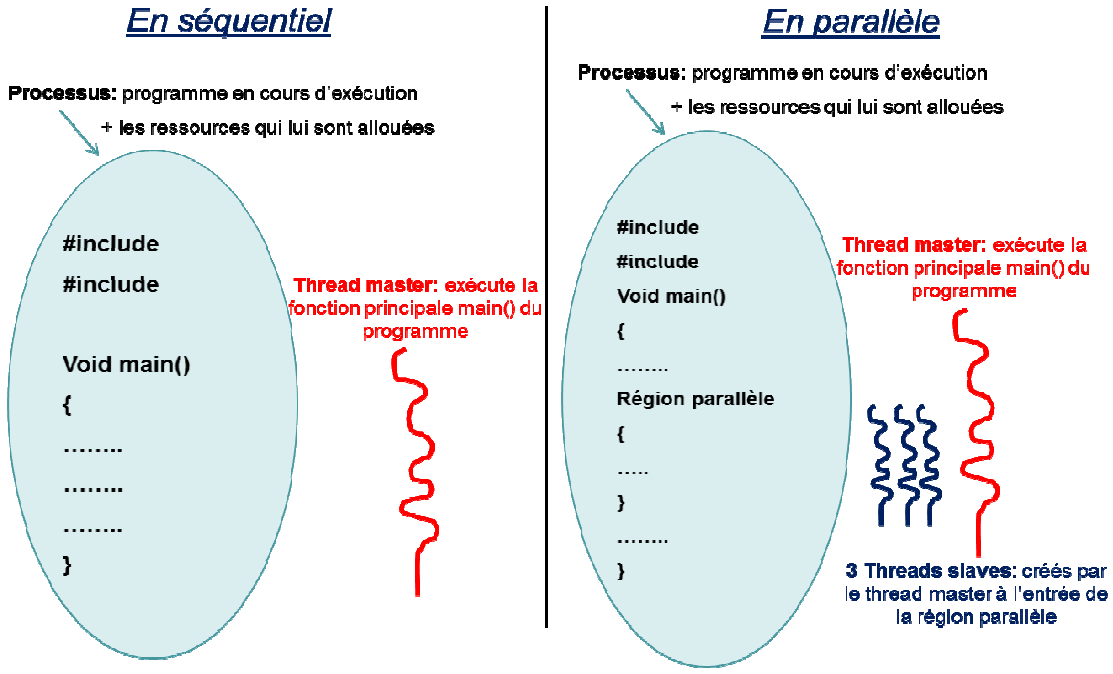


Figure 16: Modèle parallèle et séquentiel

Assignation des « threads » aux processeurs

Les threads sont affectés aux processeurs (cores d'un processeur) par le gestionnaire de tâches du système d'exploitation[8]. Ainsi OpenMP fournit un modèle de programmation de haut niveau, car il gère les threads et les processus et non pas les processeurs (voir figure 17).

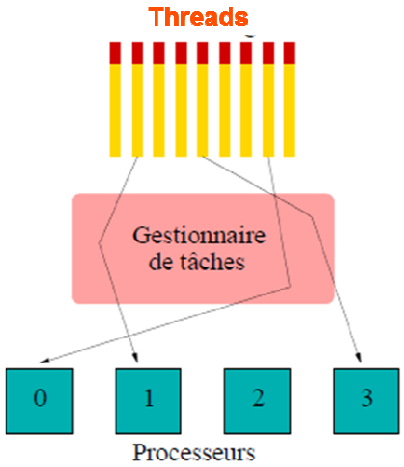


Figure 17: Assignation des threads aux processeurs

Modules d'OpenMP :

Directives et clauses de compilation :
Elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données. Elles sont considérées par le compilateur comme des lignes de commentaires à moins de spécifier une option adéquate de compilation pour qu'elles soient interprétées.
Fonctions et sous-programmes :
Ils font partie d'une bibliothèque chargée à l'édition de liens du programme.
Variables d'environnement :
Une fois positionnées, leurs valeurs sont prises en compte à l'exécution[6].

Le schéma suivant présente l'intervention des modules d'OpenMP dans les différentes phases de compilation d'un programme :

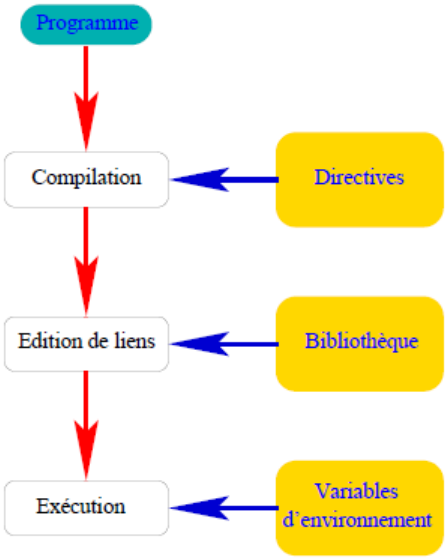


Figure 18 : Modules d'OpenMP

Structure d'Ope

OpenMP fournit un modèle de programmation basé sur les threads à un «haut niveau»
L'utilisateur n'a pas besoin de spécifier tous les détails, prend des décisions stratégiques (définir les régions qui vont être paralléliser et insère des directives de compilation pour spécifier comment ce travail va être partagé…), tandis que le compilateur se charge du reste [9].

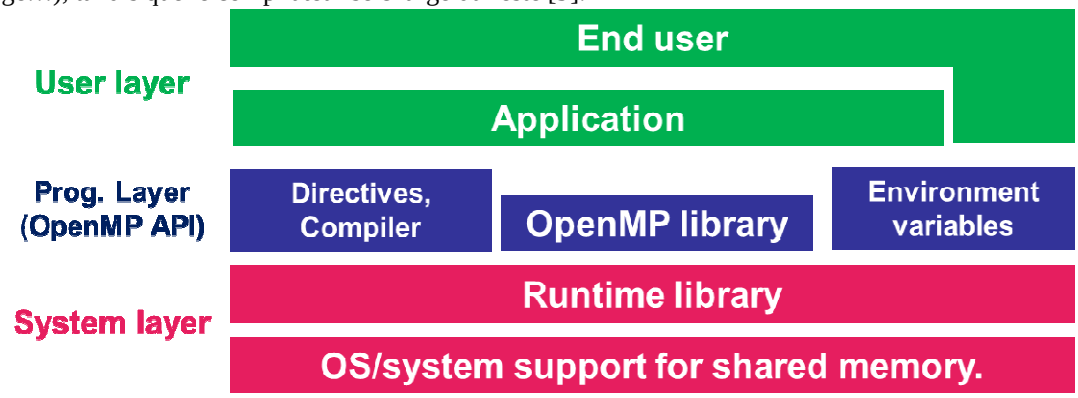


Figure 19: Structure d'OpenMP

Directives OpenMP

Une directive OpenMP possède la forme générale suivante :

sentinelle directive [clause[clause]...]

C'est une ligne qui doit être ignorée par le compilateur si l'option permettant l'interprétation des directives OpenMP n'est pas spécifiée. La sentinelle est une chaine de caractères dont la valeur dépend du langage utilisé.

En C/C++, la syntaxe générale est la suivante :

#pragma omp nom_directive [clause1 clause2 ...]

OpenMP fournit un ensemble de directives et de clauses (voir tableau 3 et 4), chacune permettant d'exprimer un besoin particulier au compilateur[10].

DIRECTIVES	INFORMATIONS
#pragma omp PARALLEL	Créer une région parallèle
#pragma omp FOR	Paralléliser une boucle for
#pragma omp SECTIONS	Paralléliser des portions du code
#pragma omp PARALLEL FOR	Créer une région parallèle et paralléliser une boucle for
#pragma omp PARALLEL SECTIONS	Créer une région parallèle et paralléliser des portions du code
#pragma omp SINGLE	Exécuter une portion de code par un seul thread uniquement
#pragma omp ATOMIC	Définir une ligne du code qui doit être exécuter par un thread à la fois
#pragma omp CRITICAL	Définir une portion du code qui doit être exécuter par un thread à la fois
#pragma omp ORDERED	Exécuter une portion du code dans une boucle for dans l'ordre des indices croissants
#pragma omp BARRIER	Définir un point de synchronisation de l'ensemble des threads dans une région parallèle.
#pragma omp FLUSH	Rafraichir la valeur d'une variable partagée en mémoire globale entre les différents threads
#pragma omp THREADPRIVATE	Dire à chaque thread de garder une même copie d'une variable privé entre différentes régions parallèles

Tableau 3: Directives d'OpenMP

CLAUSES	INFORMATIONS
IF(expression)	Spécifier une condition sur la parallélisation
PRIVATE(list)	Spécifier une liste de variables privées pour chaque thread
SHARED(list)	Spécifier une liste de variables partagées entre les threads
DEFAULT(list)	Spécifier le statut (privé ou partagé) des variables
FIRSTPRIVATE(list)	Rend une liste de variables privés et les initialise avec la valeur en mémoire
LASTPRIVATE(list)	Rend une liste de variables privés et enregistre la dernière modification dans la variable d'origine en mémoire
REDUCTION(operator:list)	Une copie privée d'une liste de variables est créé pour chaque thread, et à la fin un operator est appliqué à toutes les copies de variables
COPYIN(list)	Initialiser les variables qui ont été déclarés THREADPRIVATE à l'entrée de la première région parallèle
COPYPRIVATE(list)	Diffuser les valeurs acquises par un seul thread aux autres threads à la fin d'une construction SINGLE
SCHEDULE	Spécifie comment les itérations d'une boucle FOR vont être répartis entre les différents threads
ORDERED	Exécuter une portion du code dans une boucle FOR dans l'ordre des indices croissants
NOWAIT	Annuler la barrière de synchronisation implicite à la fin d'une construction

Tableau 4: Clauses d'OpenMP

Chaque directive supporte certaines clauses, ci-dessous un récapitulatif des combinaisons clauses/directives.

DIRECTIVES / CLAUSES	PARALLEL	FOR	SECTIONS	SINGLE	PARALLEL FOR	PARALLEL SECTIONS	TASK
IF(expression)	*				*	*	*
PRIVATE(list)	*	*	*	*	*	*	*
SHARED(list)	*	*			*	*	
DEFAULT(list)	*				*	*	*
FIRSTPRIVATE(list)	*	*	*	*	*	*	*
LASTPRIVATE(list)		*	*		*	*	
REDUCTION(operator :list)	*	*	*		*	*	
COPYIN(list)	*				*	*	
COPYPRIVATE(list)				*			
SCHEDULE		*			*		
ORDERED		*			*		
NOWAIT		*	*	*			
UNTIED							*

Tableau 5: Combinaisons clauses/directives

Remarque: pour avoir les détails de chaque directive, illustrés par des programmes exemple, voir annexe 1.

Statut d'une variable

Pendant l'exécution d'une tâche, une variable peut être lue et/ou modifiée en mémoire.
Le statut d'une variable dans une région parallèle peut être soit privé ou partagé suivant les besoins de l'application et le jugement du développeur.
Variable partagée: elle est définie dans une espace mémoire partagée, donc accessible par tous les threads.
Variable privée: elle est définie dans la pile de chaque thread, elle n'est accessible qu'à l'intérieur du thread[10].

Variable partagée

Une variable partagée est définie dans un espace mémoire partagée, donc accessible par tous les threads en lecture et écriture (voir figure 20), sa valeur à l'entrée de la région est la même pour les threads. A la sortie de la région elle garde la valeur de la dernière modification.

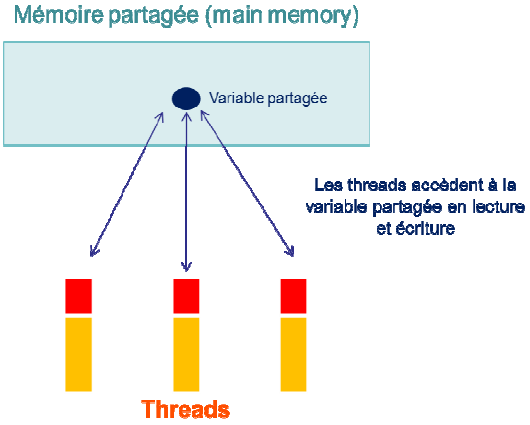


Figure 20: Variable partagée

Variable privée

Une variable privée est instancié dans la pile de chaque thread, elle n'est accessible qu'à l'intérieur du thread (voir figure 21). Elle a une valeur indéterminée à l'entrée de la région parallèle. Les modifications qu'elle va subir ne seront visible qu'à l'intérieur du thread; en effet, chaque thread dispose de sa propre copie de la variable. A la sortie de la région parallèle sa valeur est indéterminée.

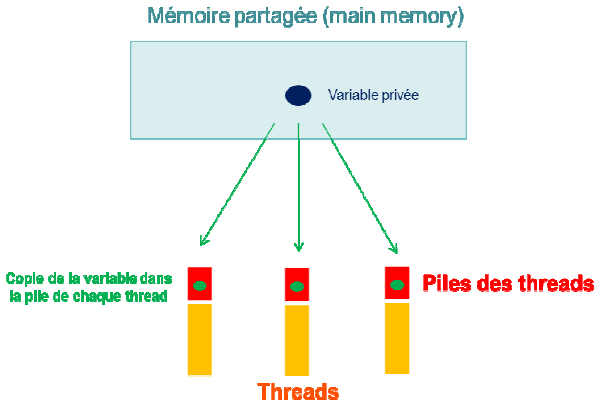


Figure 21: Variable privée

Les fonctions

La bibliothèque d'exécution ou le runtime library fournit un certain nombre de fonctions pour avoir des informations sur l'environnement d'exécution.
Quand ces fonctions sont appelées dans le programme, il faut inclure un fichier omp.h dans le code source. Ce fichier contient les prototypes de ces fonctions.

Remarque: ces fonctions peuvent changer d'une implémentation à une autre suivant la bibliothèque d'exécution utilisée. Dans ce qui suit, on présentera les principales fonctions qui existent dans la plupart des implémentations OpenMP.

Contrôle de l'environnement

Modèle d'exécution

Void omp_set_dynamic(int dynamic_threads): active/désactive l'ajustement dynamique du nombre de threads
int omp_get_dynamic(): indique si l'ajustement dynamique du nombre de threads est activée.
void omp_set_nested(int nested): active/désactive l'imbrication des régions parallèles
int omp_get_nested(): indique si l'imbrication des région parallèles est activée

Contrôle du nombre de threads

void omp_set_num_threads(int nb): Fixe le nombre de threads qu'il y aura dans la prochaine région parallèle. Elle doit être appelée à partir d'une région séquentielle.
int omp_get_num_threads(): retourne le nombre réel de threads utilisé dans la région aux moment de l'appel.
int omp_get_max_threads(): retourne le nombre maximal de threads retournée par la fonction omp_get_num_threads().
int omp_get_num_procs(): retourne le nombre de processeurs utilisés.

Contrôle manuel de la parallélisation

Exécution conditionnelle et identifiant

int omp_in_parallel() : indique si on est dans une région parallèle
int omp_get_thread_num() : retourne l'identifiant du thread courant. Très utile si l'on veut affecter une tâche à un thread particulier.

Verrouillage

Un verrou est libre ou possédé par un thread. OpenMP fournit des fonctions permettant de les manipuler, par exemple attendre à un point du code que le verrou soit libéré par un autre thread.
Ces fonctions permettent des comportements similaires à la directive critical, mais avec un contrôle plus fin.
void omp_init_lock(omp_lock_t * lock) : initialise un verrou
void omp_destroy_lock(omp_lock_t * lock) : détruit un verrou préalablement initialisé.
Void omp_set_lock(omp_lock_t * lock) : appel bloquant pour l'acquisition du verrou, si le verrou n'est pas libre l'appelant va rester bloquer jusqu'à ce le verrou soit libéré par celui qui le détient.
void omp_unset_lock(omp_lock_t * lock) : libère un verrou acquis
int omp_test_lock(omp_lock_t * lock) : appel non bloquant pour l'acquisition d'un verrou, si le verrou est acquis la fonction retourne une valeur non nulle.

Temps d'exécution

double omp_get_wtick() : retourne la précision des mesures en seconde, le nombre de seconde écoulé entre deux impulsions successives d'horloge

double omp_get_wtime() : retourne le nombre de seconde écoulé, cette valeur dépend d'un thread à un autre et varie suivant la charge du système.[10]

Les variables d'environnement

Les variables d'environnement peuvent être modifiées au moment de l'exécution. Il en existe qui reproduisent le même travail que les fonctions de la bibliothèque. Or leur avantage est qu'on peut modifier l'environnement d'exécution sans avoir à modifier le code source et le recompiler. Donc, idéal pour un utilisateur final de l'application.

Remarque: ces variables peuvent aussi changer d'une implémentation à une autre suivant la bibliothèque d'exécution utilisée. Dans ce qui suit, on présentera les principales variables d'environnement qui existent dans la plupart des implémentations OpenMP.

- OMP_DYNAMIC : active/désactive l'ajustement dynamique du nombre des threads.
- OMP_NUM_THREADS : fixe le nombre maximal de threads à utiliser
- OMP_NESTED : active/désactive l'imbrication des régions parallèles
- OMP_SCHEDULE : définit la stratégie de distribution des itérations des boucles dont la clause schedule est fixée à runtime.
- OMP_STACKSIZE : contrôle la taille des piles d'exécution des threads.
- OMP_WAIT_POLICY : spécifie le comportement des threads quand ils sont en attente. Statut ACTIVE veut dire que les threads doivent consommer les cycles du processeur pendant leur attente, contrairement au statut PASSIVE.
- OMP_MAX_ACTIVE_LEVELS : spécifie le nombre maximum de régions parallèles imbriqués
- OMP_THREAD_LIMIT : spécifie le nombre de threads à utiliser pour tous le programme OpenMP.[10]

Conclusion

Dans ce chapitre, on a présenté OpenMP et les différents modules qu'il propose. Dans le chapitre suivant, on va l'appliquer à la chaine radar en proposant des modèles de parallélisation.

Dans ce chapitre, on procèdera par une description générale du système radar, ses composants et ses données de base, puis une description de chaque bloc de la partie traitement cohérent de la chaine radar. Ensuite, l'implémentation Matlab qui est une étape nécessaire avant l'implémentation en C. Enfin, l'implémentation OpenMP et les différents scénarios proposés.

Description générale du système radar

Chapitre 4:ApplicationOpenMP sur la chaine radar

Le Radar (**RA**dio **D**etection **A**nd **R**anging) est un système qui permet la détection et la mesure des caractéristiques (distance, vitesse.. .) d'un objet au moyen d'un rayonnement électromagnétique réfléchi par celui-ci.

Un radar émet de puissantes ondes, produites par un oscillateurradio et transmises par une antenne (voir figure 22). Bien que la puissance des ondes émises soit grande, l'amplitude du signal renvoyé est le plus souvent très petite. Néanmoins, les signaux radio sont facilement détectables et peuvent être amplifiés de nombreuses fois.

En analysant le signal réfléchi, il est possible de localiser et d'identifier l'objet responsable de la réflexion, ainsi que de calculer sa vitesse de déplacement. Le radar peut détecter des objets ayant une large gamme de propriétés réflectives, alors que les autres types de signaux, tels que le son ou la lumière visible, revenant de ces objets, seraient trop faibles pour être détectés. De plus, les ondes radio peuvent se propager avec une faible atténuation à travers l'air et divers obstacles, tels les nuages, le brouillard ou la fumée, qui absorbent rapidement un signal lumineux. Cela rend possible la détection et le pistage dans des conditions qui paralysent les autres technologies[11].

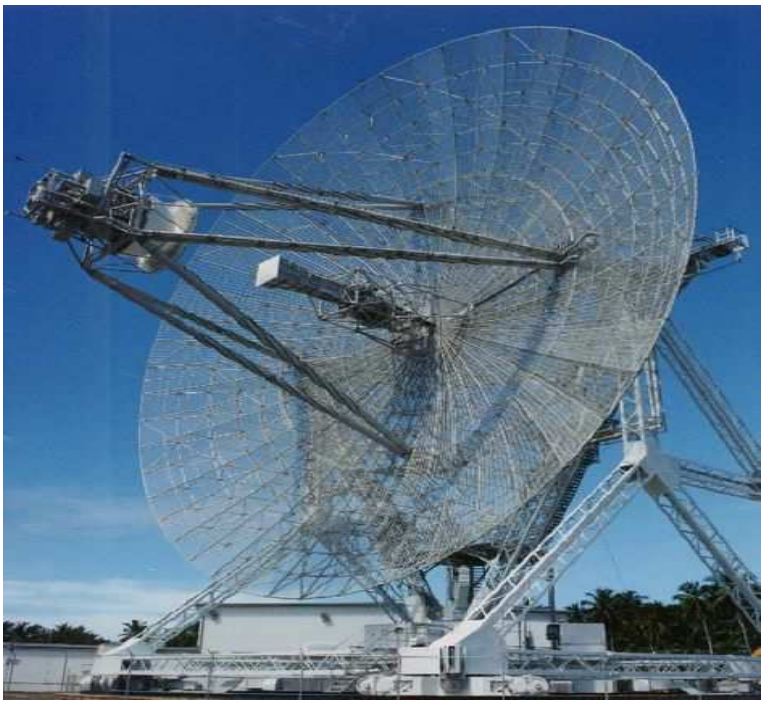


Figure 22: Antenne d'un radar

Classification des systèmes Radar

En fonction des informations qu'ils doivent fournir, les équipements radars utilisent des qualités et des technologies différentes[11]. Ceci se traduit par une première classification des systèmes radars:

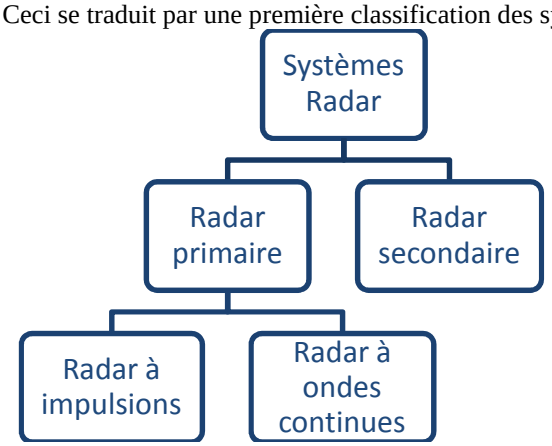


Figure 23: Différents systèmes Radar

Par la suite, on ne s'intéresse qu'au radar à impulsions.

Le radar à impulsions émet des impulsions de signal hyperfréquence à forte puissance. Chaque impulsion est suivie d'un temps de silence plus long que l'impulsion elle-même, temps durant lequel les échos de cette impulsion peuvent être reçus avant qu'une nouvelle impulsion ne soit émise. Direction,distance et parfois, si cela est nécessaire, hauteur ou

altitude de la cible, peuvent être déterminées à partir des mesures de la position de l'antenne et du temps de propagation de l'impulsion émise.

Les composantes d'un système Radar

Un radar est formé de différentes composantes:

- *L'émetteur* qui génère l'onde radio. Sur les radars à hyperfréquences (fréquences supérieures au gigahertz), c'est un guide d'onde qui amène l'onde vers l'antenne.
- *Le duplexeur*, un commutateur électronique, dirige l'onde vers l'antenne lors de l'émission ou le signal de retour depuis l'antenne vers le récepteur lors de la réception quand on utilise un radar mono statique. Il permet donc d'utiliser la même antenne pour les deux fonctions.
- *L'antenne*, dont le rôle est de diffuser l'onde électromagnétique vers la cible avec le minimum de perte. Sa vitesse de déplacement, rotation et/ou balancement, ainsi que sa position, en élévation comme en azimut, sont asservies, soit mécaniquement, mais parfois aussi électroniquement. L'antenne est sollicitée à l'émission et à la réception. Ces deux fonctions peuvent être cependant séparées entre deux antennes dans le cas de radars multistatiques.
- *Lerécepteur* qui reçoit le signal incident (cible - antenne - guide d'ondes - duplexeur), le fait émerger des bruits parasites, l'amplifie et le traite.
- *Un étage de traitement de signal* permettant de traiter le signal brut afin d'en extraire des données utiles à l'opérateur (détection, suivi et identification de cible; extraction de paramètres météorologiques, océanographiques, etc.)

Données de base du traitement Radar

- **Les pulses :**

L'impulsion radar est un nombre bien défini d'échantillons qui sont produites à partir de l'onde continue générée par l'oscillateur via des générateurs d'impulsions, ou modulateurs. Ils laissent passer l'onde vers l'amplificateur durant un très court laps de temps (de l'ordre de la µ seconde) ce qui permet de concentrer l'énergie de l'onde dans cette impulsion (puissance de l'ordre du MWatt).

- **Les cases distances :**

Une Case Distance correspond à la période d'échantillonnage, CD= 1/Fe (Fe : Fréquence d'échantillonnage du signal)

- **Les voies :**

Une voie correspond à une antenne élémentaire. Un Radar peut contenir : 8, 16, 32 ou 64 voies. Il faut noter que le traitement Radar est réalisé sur une succession de bursts indépendantes sous des contraintes techniques consistant généralement à ne pas faire de réception pendant l'émission.

- **Les beams :**

Elle correspond à une sortie élémentaire. Dans notre cas on travaille sur 16 beams.
La figure ci-dessous illustre la représentation des différentes données de base du traitement radar:

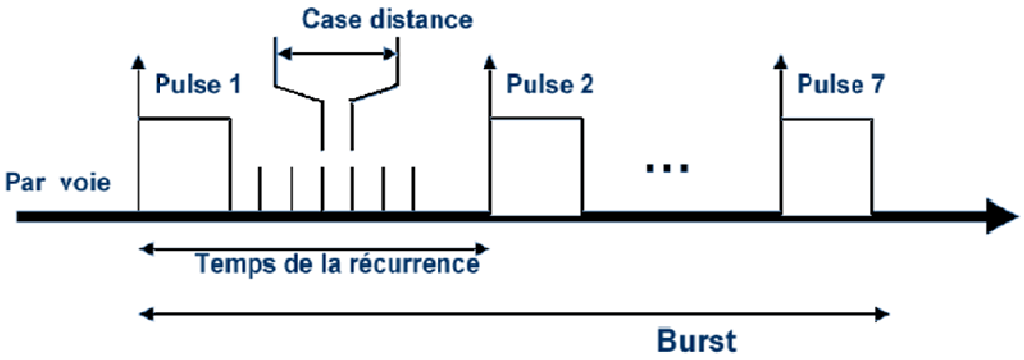


Figure 24: Représentation des données de base de traitements radar

La figure ci-dessous illustre la représentation des trois axes indépendants du traitement :

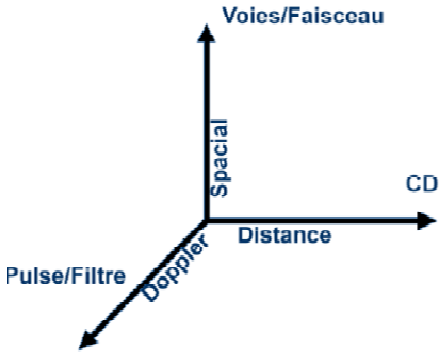
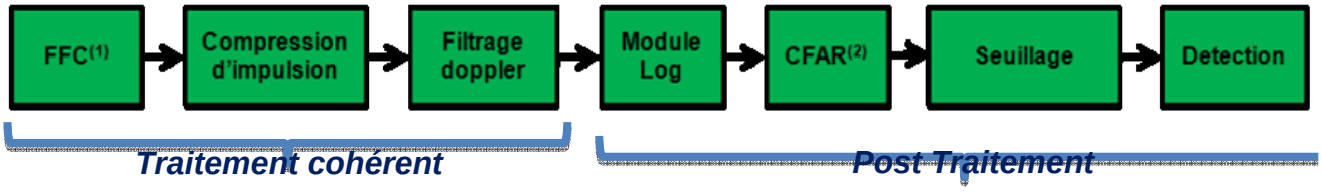


Figure 25: Axes du signal Radar

Les algorithmes de traitement Radar

La chaîne de traitement Radar se compose de différents algorithmes de traitement de signal qui sont présentés comme suit :



(1) : Formation de faisceaux par le calcul
(2) : Constant false alarm rate

La chaîne de traitement radar peut être séparée en deux blocs[11] :

- Les étapes de traitement cohérent qui visent à améliorer le rapport signal sur bruit.
- Le post-traitement qui est simplement une prise de décision, fonction des performances souhaitées.

Notre mission porte uniquement sur la chaine du traitement cohérent dont les étapes sont détaillées ci-dessous :

La formation de faisceaux par le calcul

Les radars à antennes mécaniques, ne pouvant plus répondre aux exigences de performance et de fiabilité croissantes, ont favorisé l'apparition d'une autre classe d'antennes : les antennes à réseaux de capteurs, permettant de s'affranchir des éléments mécaniques en les remplaçant par des méthodes calculatoires.

Leur principe de fonctionnement est la création d'un diagramme de rayonnement modifiable par traitement des signaux reçus sur les différents capteurs. Ce traitement peut s'effectuer par l'introduction de déphasages ou en influant sur l'amplitude des signaux reçus sur chaque capteur. Par cette méthode on peut focaliser l'énergie sur certaines directions qu'on privilégie, ce qui reviendrait dans le cas du radar à antenne mécanique à tourner l'antenne[12].

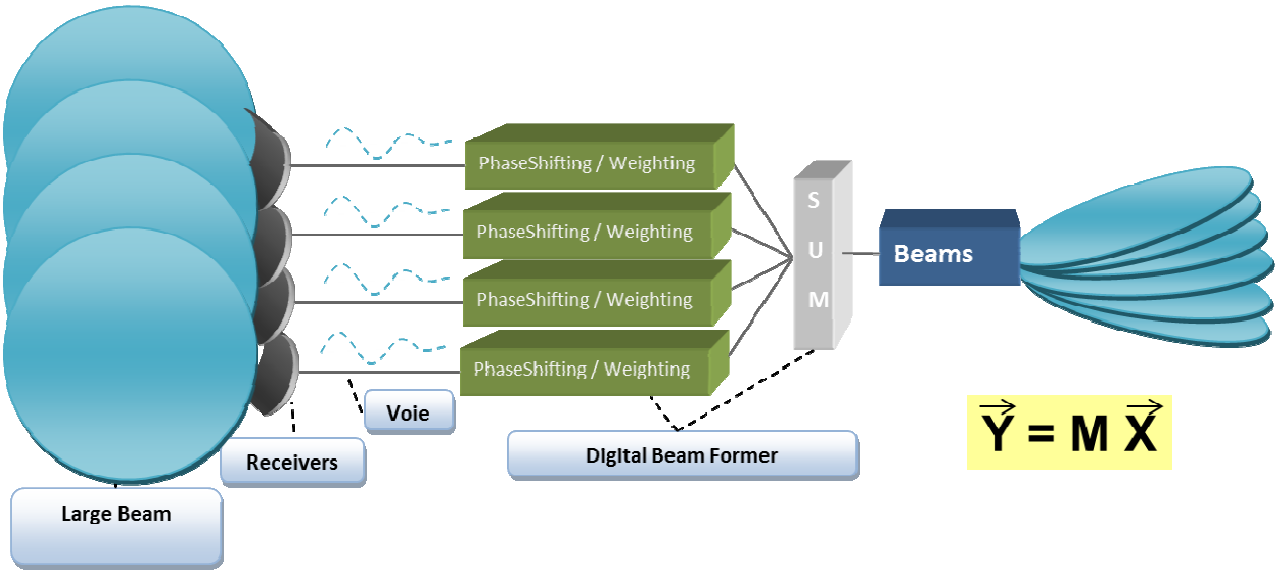


Figure 26: Schématisation de la formation de faisceaux par le calcul

La FFC est donc une technique générale de traitement de signal utilisée pour contrôler la direction de réception ou de transmission d'un signal. En utilisant la FFC, il est possible de :

- Diriger la majorité de l'énergie du signal à transmettre dans une direction angulaire donnée.
- Calibrer un groupe de transducteurs lors de la réception des signaux afin de prédominer la réception à partir d'une direction angulaire donnée.

Remarque : Lors de notre étude le nombre de faisceaux utilisés est inférieur au nombre de voies, la puissance de calcul est ainsi réduite si la formation de faisceaux par le calcul est la première étape de la chaîne de traitement radar.

La Compression d'impulsion

La compression d'impulsion est utilisée afin d'augmenter la résolution en distance de la mesure radar, ainsi que le rapport signal sur bruit, par modulation en fréquence du signal émis et par filtrage adapté en réception[12].

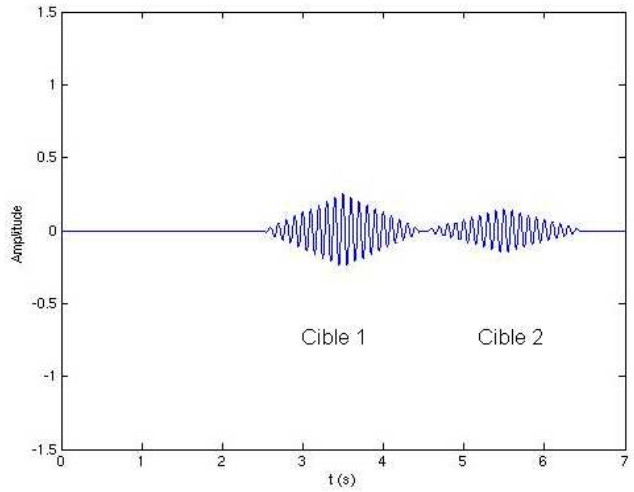


Figure 27: Signal après filtrage adapté sans modulation de fréquence en entrée

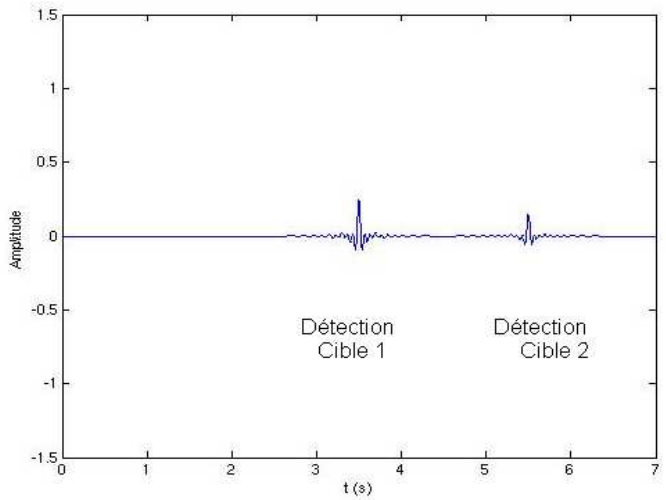


Figure 28: Signal après filtrage adapté avec modulation de fréquence en entrée

On observe bien qu'après filtrage adapté avec modulation de fréquence en entrée, il est plus facile de distinguer deux cibles très proches (figure 28). Sans modulation du signal émis, la trace des cibles est plus diffuse (figure 27).

Remarque : Comme les cibles traquées ne sont que très rarement isolées dans l'espace, il convient de réaliser le traitement « doppler » et « compression d'impulsion » avant de comparer le signal à un seuil qui décide si l'écho reçu provient ou non d'une cible. Cette étape est réalisée par un algorithme appelé CFAR.

Le filtrage doppler

Après émission, l'onde rayonnée par l'antenne du radar heurte de nombreux obstacles naturels : végétations, reliefs, nuages, pluie….
Chaque élément d'obstacle crée un écho que capte l'antenne. La superposition de ces échos produit un signal indésirable, beaucoup plus puissant que le signal recherché, issu des cibles. Pour distinguer l'écho utile des nuisibles, le radar exploite le fait que les cibles se déplacent. Il émet plusieurs impulsions identiques et compare les signaux reçus. S'ils viennent de réflecteurs fixes (parasites) les signaux seront semblables, s'ils viennent de réflecteurs mobiles (cibles) les signaux varieront (modulation de fréquence)[12].

On suppose que les éléments réflecteurs dans une direction donnée sont tous fixes, hormis une cible mouvante. On émet deux impulsions identiques.

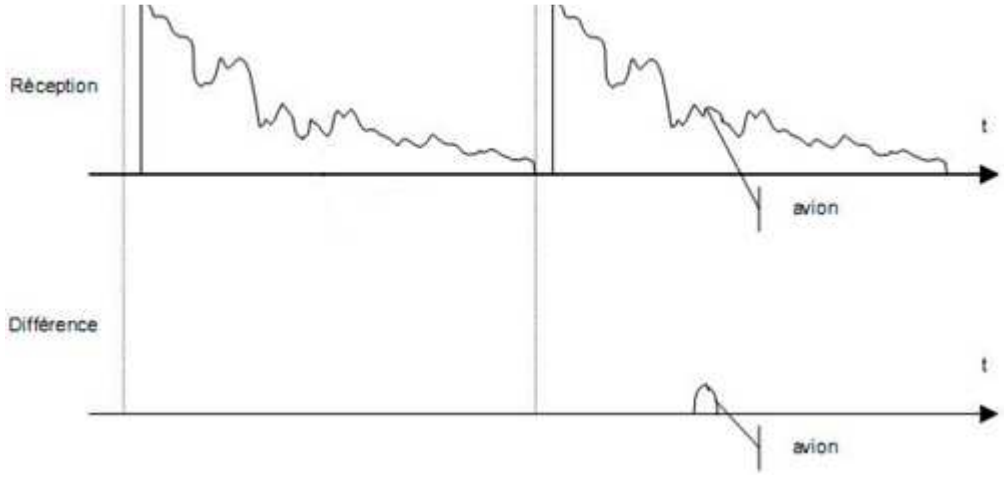


Figure 29:Visualisation du traitement minimal pour filtrage doppler

Le traitement minimal qui permet de gommer le paysage des réflecteurs fixes tout en conservant la visibilité de la cible est une simple soustraction des échantillons de deux impulsions successives

Implémentation Matlab

Avant de commencer l'implémentation des algorithmes sous Visual Studio (VS) en langage C et OpenMP, la phase de développement Matlab est nécessaire. En ce sens, elle permet la création des scénarios de référence, avec d'une part les fichiers utiles pour la simulation du signal d'entrée, et d'autre part des fichiers pour la validation des algorithmes sur la cible.
De ce fait, nous n'effectuerons que le traitement cohérent dans cette étude, qui donne des résultats uniques et donc qui pourront valider par comparaison de fichiers les algorithmes sous VS.
De plus, d'un point de vue formateur, Matlab fut un bon moyen pour comprendre les algorithmes de la chaine de traitement du signal radar.

Première approche des algorithmes du traitement RADAR

Lorsqu'un radar détecte une cible, cela signifie qu'il possède trois informations essentielles sur cette dernière :

- Sa vitesse ou fréquence doppler
- Son éloignement ou distance au radar
- Son altitude ou fréquence spatiale

Nous allons donc travailler dans un espace à trois dimensions (voir figure 30), où chacune d'entre elles renseigne sur une des caractéristiques de la cible.

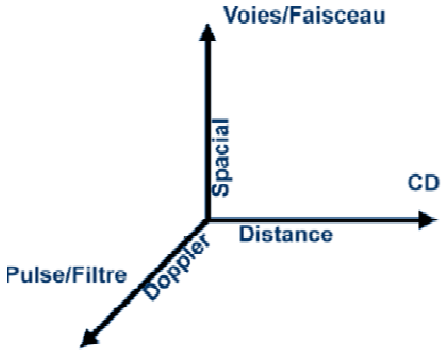


Figure 30: Dimensions d'un signal radar

Les paramètres dont nou

- N_r = 2000 Cases distance
- N_s = 64 voies
- N_b= 16 faisceaux
- N_p = 16 pulses
- N_d = 16 filtres doppler

Chacun de ces paramètres interviendra dans l'un des trois modules du traitement cohérent.

Réalisation du signal d'entrée de référence

Nous pouvons écrire le signal d'entrée comme une variable, d'un espace à trois dimensions, modélisée comme suit :

$$Input(n_v, n_p, n_r) = \begin{cases} \rho * e^{i\varphi} * e^{\frac{2i\pi f_v n_v}{N_v}} * e^{\frac{2i\pi f_p n_p}{N_p}} * e^{\frac{2i\pi n_r^2}{\tau}} & pour\ n_p \in [\![N_{p1}, N_{p2}[\![\quad et\ n_r \in [\![d, d+101[\![\\ 0 & pour\ n_p \in [\![0, N_{p1}[\![\quad ou\ n_r \notin [\![d, d+101[\![\end{cases}$$

ρ : $l'amplitude\ du\ signal$
φ : $sa\ phase$
f_v : $la\ fréquence\ spatiale$
f_d: $la\ fréquence\ doppler$
d : $la\ distance\ de\ la\ cible\ au\ radar$
$n_v \in [\![0, N_v[\![\quad n_p \in [\![0, N_p[\![\quad n_r \in [\![0, N_r[\![$

Chaque cible étant modélisée par un signal de forme décrite ci-dessus, dans l'hypothèse ou plusieurs cibles devraient être modélisées le signal d'entrée global sera la somme des signaux générés par chacune d'entre elles.

Afin de permettre au modèle de coller d'avantage à la réalité pour notre modélisation Matlab nous avons ajouté un bruit blanc gaussien au signal d'entrée. Ce bruit est modélisé par une matrice à trois dimensions contenant des valeurs distribuées suivant la loi normale avec une moyenne nulle et un écart type de 1.

Constitution des jeux d'essais fonctionnels de référence

Afin de valider notre implémentation sur VS en langage C de la chaîne de traitement radar, nous avons besoin d'éditer des jeux d'essais fonctionnels de référence. Ces derniers auront trois fonctions :

- Créer une matrice d'entrée pour l'implémentation en langage C
- Valider la création des jeux de coefficients
- Valider, d'abord chaque étape, puis la chaîne complète de traitement radar.

Pour chacune de ces fonctions un fichier de données est créé. La matrice d'entrée sera utilisée comme point de départ du programme en C. Pour les étapes de validation les valeurs de ces fichiers seront comparées aux valeurs générées sous VS.

Architecture complète du programme Matlab

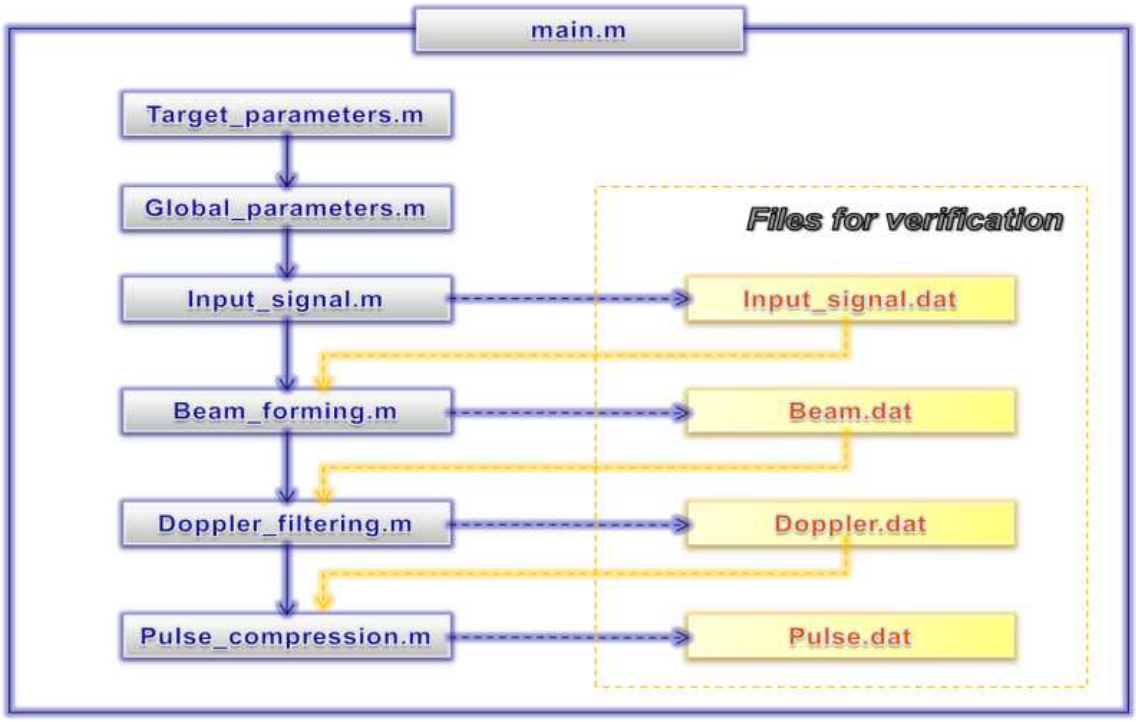


Figure 31: Architecture du programme Matlab

En bleu sont les différents fichiers créés sous Matlab avec un main, deux fichiers pour les paramètres et quatre fichiers pour les différents algorithmes.
En jaune, on a d'abord l'input_signal.dat qui nous donnera les valeurs d'entrées. Ainsi, avec le même signal d'entrée sous Matlab et sous VS, on pourra vérifier par comparaison de fichiers (dont les trois autres encadrés jaunes) avec les prochains algorithmes du traitement cohérent implémentés en C.

Implémentation en C

Le présent chapitre présentera l'implémentation des différents algorithmes de la chaine du traitement cohérent en langage C sous VS pour pouvoir y intégrer OpenMP par la suite.
Le signal d'entrée de référence généré par Matlab sera aussi le fichier d'entrée de l'implémentation en C. Aussi, l'étude Matlab va permettre de valider les fichiers de sorties générés par VS.

Architecture du projet

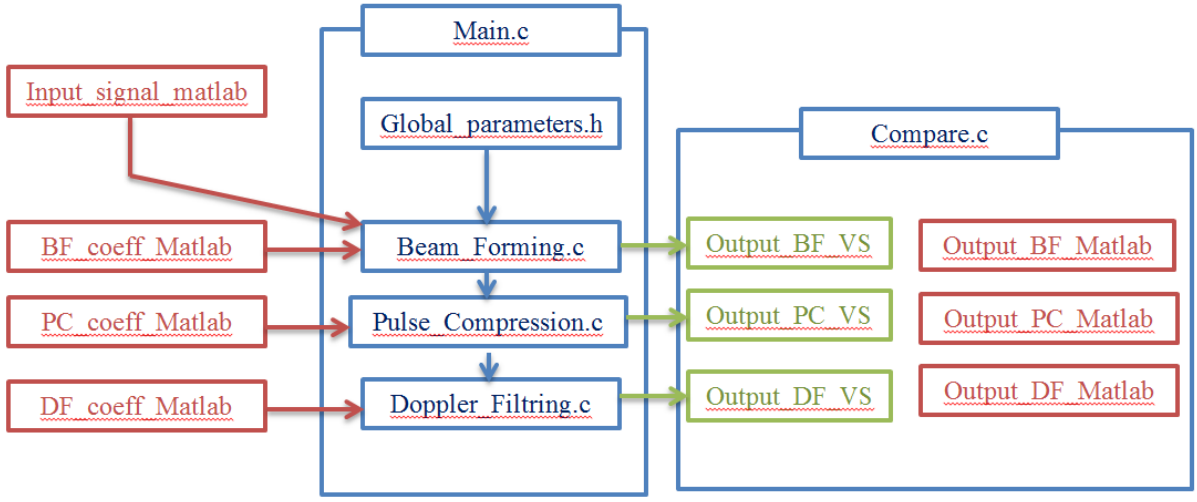


Figure 32: Architecture de l'implémentation en C

Le signal d'entrée généré par Matlab, qui représente le signal reçu par le radar est injectée dans le projet Visual Studio. Ainsi que les coefficients du calcul nécessaires pour chaque traitement.
Pour valider les résultats obtenus on compare la sortie de VS avec la sortie du Matlab.

Les fichiers Beam_forming.c, pulse_compression.c et doppler_filtring.c représentent respectivement le code C des algorithmes de la formation de faisceaux par le calcul (FFC), la compression d'impulsions et le filtrage doppler.

Nous présenterons ci-après quelques lignes de code de ces trois blocs,pour donner une idée sur le traitement et aussi afin d'y ajouter dans le prochain chapitreles modules d'OpenMP pour la parallélisation.

Code C de la formation de faisceaux par le calcul

```
void Beam_Forming (float *Input, float *coeff_BF, float *Output_BF)
{
	float Real,Imag;
	int np,nr,nb,ns;
	int i=0,k=0,j;
	int aux_i;
	int reg[Np*2];
	for(i=0;i<Np*2;i+=2)
	{
```

```
                reg[i]=Ns*Np*Nr*2*i/(Np*2);
                reg[i+1]=Nb*Np*Nr*2*i/(Np*2);
        }
        for(np=0 ; np<Np ; np++)
        {
                i=reg[2*np];
                k=reg[2*np+1];
                for(nr=0;nr<Nr;nr++)
                {
                        // code et traitement
                }
        }
}
```

Code C de la compression d'impulsion

```
void pulse_compression (float *Input_CI, float *CI_Coeff, float *Output_CI)
{
        float Real,Imag;
        int np,nr,nb;
        int coef,a,i;

        for(np=0;np<Np;np++)
        {
                for(nb=0;nb<Nb;nb++)
                {
                        // code et traitement
                 }
        }
}
```

Code C du filtrage doppler

```
void Doppler_Filtring (float *Input_DP, float *DP_Coeff, float *Output_DP)
{

        float Real,Imag;
        int np,nr,nb,nd;
        int i,j,k;

        for(nb=0;nb<Nb;nb++)
        {
                for(nr=0 ;nr<Nr;nr++)
                {
                        // code et traitement
                }
        }
}
```

Ce code écrit en langage C est constitué principalement d'imbrications de boucles « FOR ». Ces boucles « FOR »vont faire l'objet de parallélisme dans le prochain paragraphe.

Variantes d'implémentation d'OpenMP sur la chaine radar

Nous souhaitons maintenant intégrer OpenMP dans la chaine radar qui sera implémentée sur des processeurs multi-coeurs.

Afin d'exploiter au maximun la puissance de ces processeurs, on veut paralléliser le code de la chaine radar pour qu'il soit exécuté simultanément sur les différents cores disponibles. On réduiraainsi le temps d'exécution du code par rapport à une exécution séquentielle décrite dans le chapitre précédent.

Le présent paragraphe contiendra quelques scénarios de parallélisme qu'on peut implémenter avec OpenMP sur la chaine radar. Ensuite on présentera une description détaillée de chaque cas en analysant les résultats qu'il fournit. Enfin, une conclusion et des recommandations d'implémentation.

Le processeur utilisé pour ces tests est le Intel Core2Duo, nous allons donc paralléliser le code de la chaine radar sur les 2 cores de ce processeur.

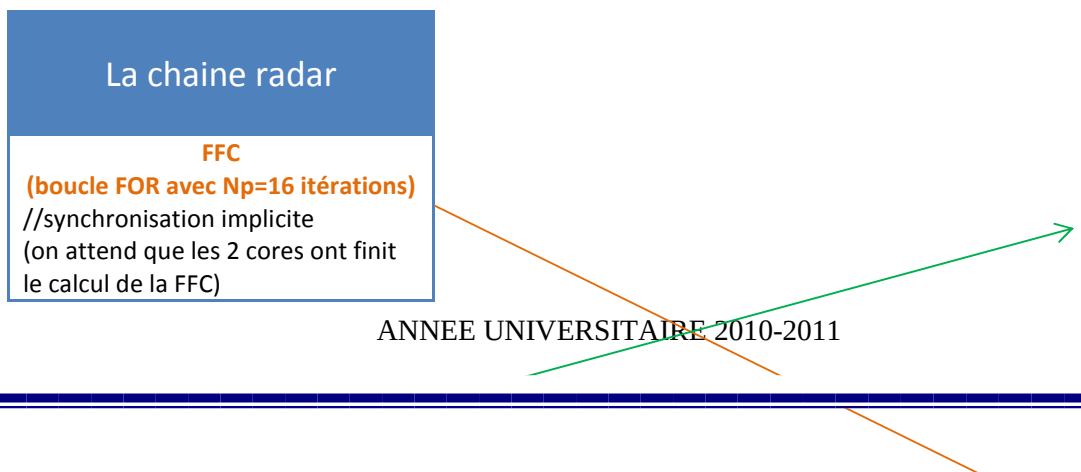
Description des cas d'implémentation

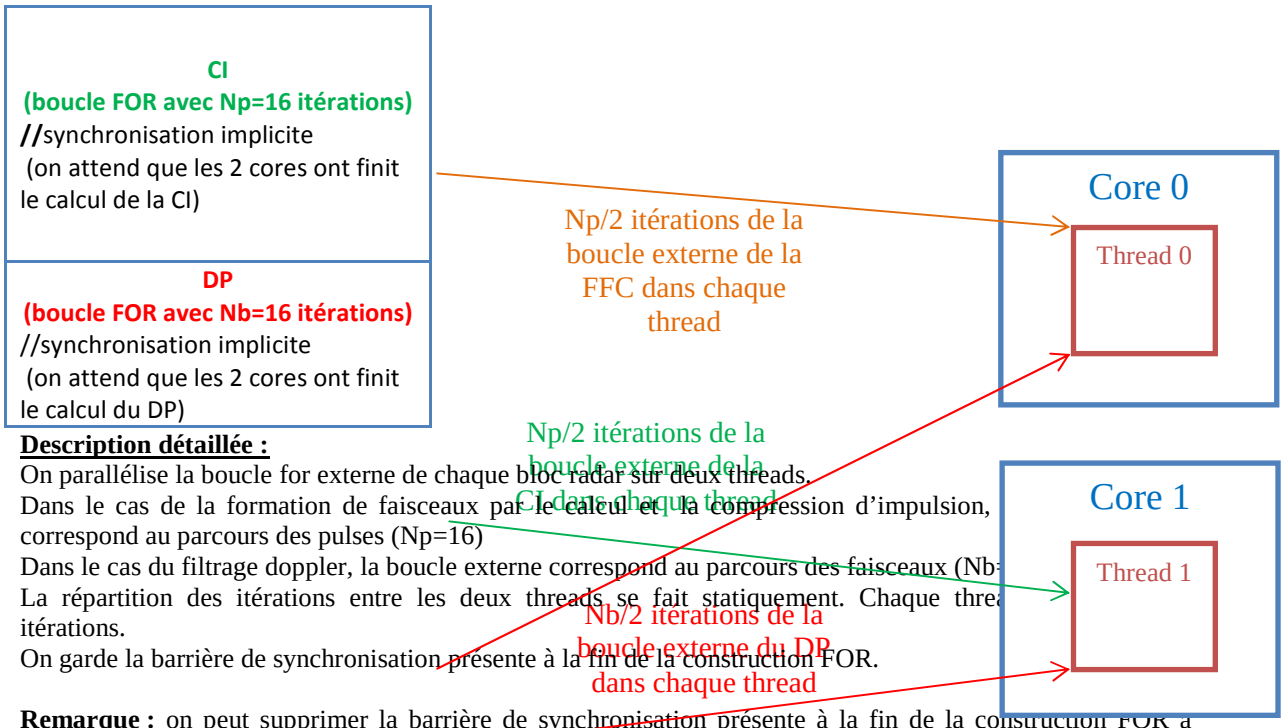
Parallélisme appliqué sur des boucles « FOR »

Parallélisme statique

On parallélise statiquement la boucle FOR externe de chaque bloc de la chaine sur les 2 cores en utilisant deux threads.

Schéma de parallélisme :





Description détaillée :
On parallélise la boucle for externe de chaque bloc radar sur deux threads.
Dans le cas de la formation de faisceaux par le calcul et la compression d'impulsion, correspond au parcours des pulses (Np=16)
Dans le cas du filtrage doppler, la boucle externe correspond au parcours des faisceaux (Nb=
La répartition des itérations entre les deux threads se fait statiquement. Chaque thread itérations.
On garde la barrière de synchronisation présente à la fin de la construction FOR.

Remarque : on peut supprimer la barrière de synchronisation présente à la fin de la construction FOR à l'aide la clause NOWAIT mais la barrière de la fin de la région parallèle sera toujours présente et ne peut pas être supprimée. Ceci ne changera rien aux performances de l'application car après la fin de la construction FOR vient directement la fin de la région parallèle, en d'autres termes avoir une seule barrière ou deux c'est la même chose.

Figure 33 : Parallélisme statique

Directives OpenMP utilisées :

#pragma omp parallel : pour créer les trois régions parallèles, chacune correspond à un bloc de la chaine radar.
#pragma omp for schedule(static) : utilisée à l'intérieur de chaque région pour paralléliser une boucle for. On utilise une répartition statique des itérations de la boucle entre les deux threads.

Parallélisme dynamique

On parallélise dynamiquement la boucle for externe de chaque bloc de la chaine sur les 2 cores en utilisant deux threads. On choisira des paquets de Np/4 itérations. Dans notre cas Np=16, les paquets seront constitués de 4 itérations.

Schéma de parallélisme :

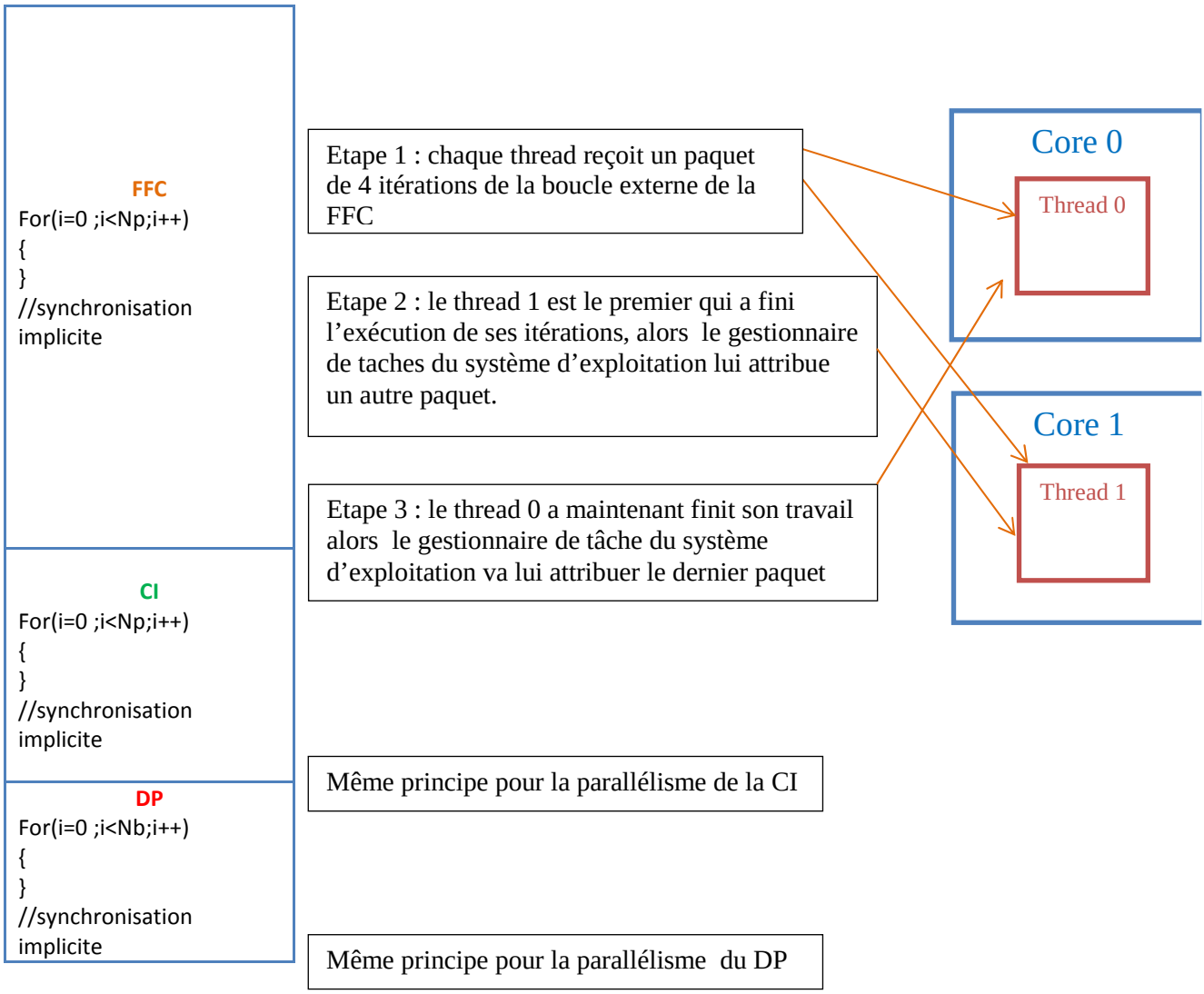


Figure 34 : Parallélisme dynamique

Description détaillée :

On parallélise toujours la boucle for externe de chaque bloc radar sur deux threads.
La répartition des itérations entre les deux threads se fait dynamiquement avec des paquets de 4 itérations. Chaque thread va recevoir 4 itérations à exécuter ensuite le premier thread qui finira son travail, le gestionnaire des tâches va lui attribuer un autre paquet de 4 itérations et ainsi de suite jusqu'à l'épuisement des paquets.
On garde la barrière de synchronisation présente à la fin de la construction FOR.

Directives OpenMP utilisées :

#pragma omp parallel : pour créer les trois régions parallèles, chacune correspond à un bloc de la chaine radar.
#pragma omp for schedule(dynamic,Np/4) : utilisée à l'intérieur de chaque région pour paralléliser une boucle for. On utilise une répartition dynamique avec des paquets de Np/4 itérations de la boucle entre les deux threads.

Parallélisme guidé

On utilise un parallélisme guidée de la boucle for externe de chaque bloc de la chaine sur les 2 cores en utilisant deux threads. On rappelle que la répartition guidée suit le même principe que la répartition dynamique, sauf que la taille des paquets n'est pas uniforme mais elle décroit exponentiellement. Les petits paquets seront constitués de Np/5 itérations.
Dans notre cas Np=16, donc Np/5=3
Paquet 1 contient 6 itérations
Paquet 2 contient 4 itérations
Paquet 3 contient 3 itérations
Paquet 4 contient 3 itérations

Schéma de parallélisme :

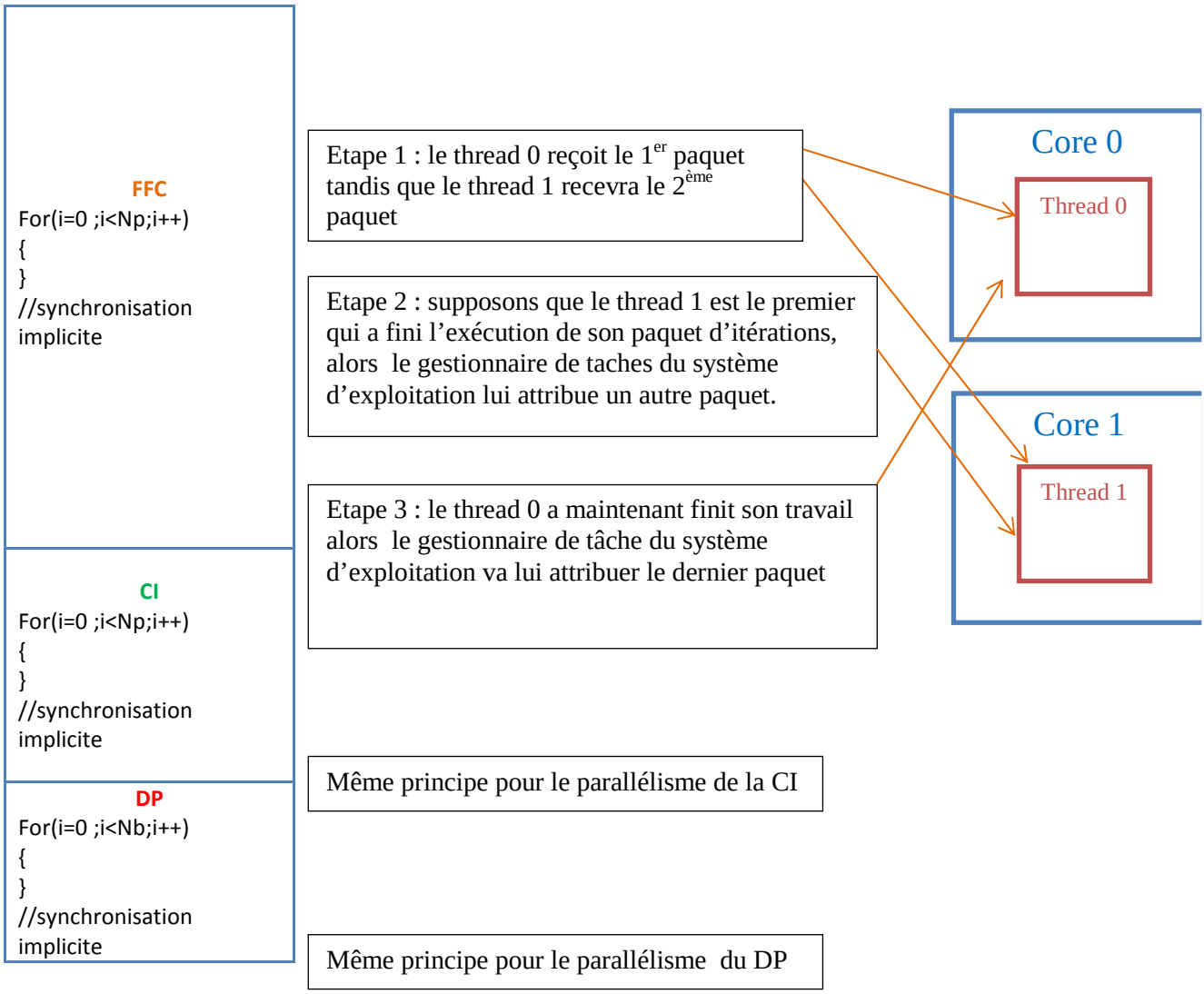


Figure 35 : Parallélisme guidé

Description détaillée :

On parallélise toujours la boucle for externe de chaque bloc radar sur deux threads.
La répartition des itérations entre les deux threads se fait d'une façon guidée avec des paquets d'itérations qui décroissent exponentiellement, le plus petit paquet contiendra 3 itérations. Chaque thread va recevoir un paquet qui contient un certain nombre d'itérations à exécuter ensuite le premier thread qui finira son travail, le gestionnaire des tâches va lui attribuer un autre paquet et ainsi de suite jusqu'à l'épuisement des paquets.
On garde la barrière de synchronisation présente à la fin de la construction FOR.

Directives OpenMP utilisées :

#pragma omp parallel : pour créer les trois régions parallèles, chacune correspond à un bloc de la chaine radar.
#pragma omp for schedule(guided,Np/5) : utilisée à l'intérieur de chaque région pour paralléliser une boucle for. On utilise une répartition guidée avec des paquets de taille décroissante dont le petit aura Np/5 itérations.

Parallélisme appliqué sur des sections du code

Le principal but de cette implémentation est de visualiser l'impact de la synchronisation dans la fin des constructions OpenMP. Pour se faire on réalisera un modèle de parallélisme qu'on testera avec et sans synchronisation.

Parallélisme en sections du code avec synchronisation

Dans une région parallèle, on va créer deux constructions, une construction FOR et une construction SECTIONS. On va garder la barrière de synchronisation à la fin de la boucle FOR.

Schéma de parallélisme :

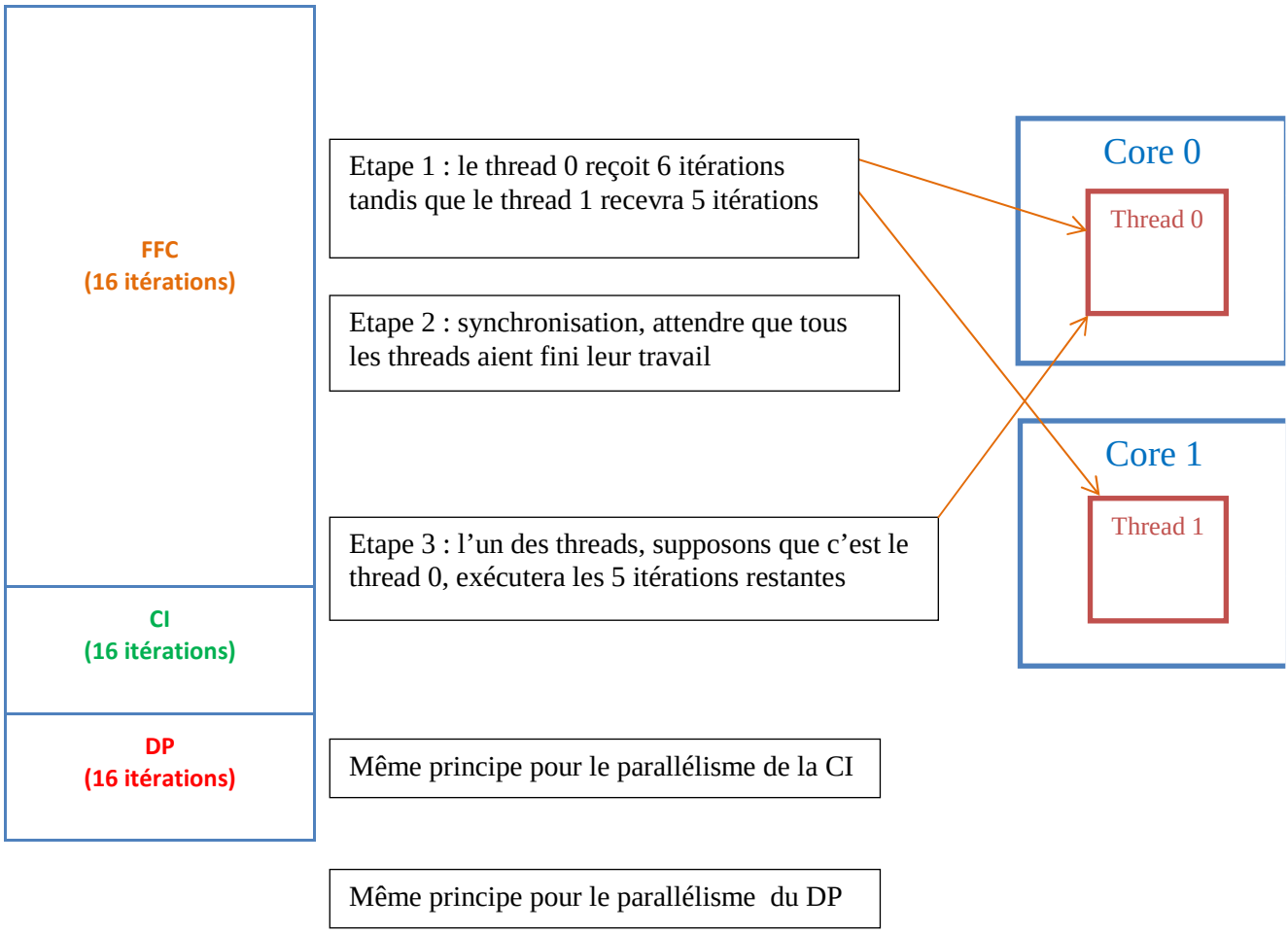


Figure 36 : Parallélisme avec synchronisation

Description détaillée :

Dans une région parallèle, on va créer deux constructions, une construction FOR et une construction SECTIONS.
Dans la construction FOR, on mettra que 11 itérations avec une répartition guidée, cela dit qu'un thread exécutera 6 itérations et l'autre exécutera 5 itérations. On garde la barrière de synchronisation présente à la fin de la construction FOR.
Dans la construction SECTIONS, on mettra alors les 5 itérations qui restent.
On aura ainsi exécutés les 16 itérations qu'on avait dans le cas séquentiel.

Directives OpenMP utilisées :

#pragma omp parallel : pour créer les trois régions parallèles, chacune correspond à un bloc de la chaine radar.
#pragma omp for schedule(guided,5) : utilisée à l'intérieur de la région parallèle pour paralléliser une boucle for allant de 0 à 11. Càd qu'un thread exécutera 6 itérations alors que l'autre exécutera 5 itérations.
#pragma omp for sections : sert à informer le compilateur de la présence d'une section du code à paralléliser par la suite
#pragma omp for section : sert à déclarer la section du code qui va être associée à un thread. Dans notre cas, on n'utilise qu'une seule section pour qu'elle soit exécutée avec un seul thread.

Université Sidi Mohamed Ben Abdellah
Faculté des Sciences et Techniques Fès
Département Génie Electrique

Parallélisme en sections du code sans synchronisation

Dans une région parallèle, on va créer deux constructions, une construction FOR et une construction SECTIONS. On va supprimer la barrière de synchronisation à la fin de la boucle FOR.

Schéma de parallélisme :

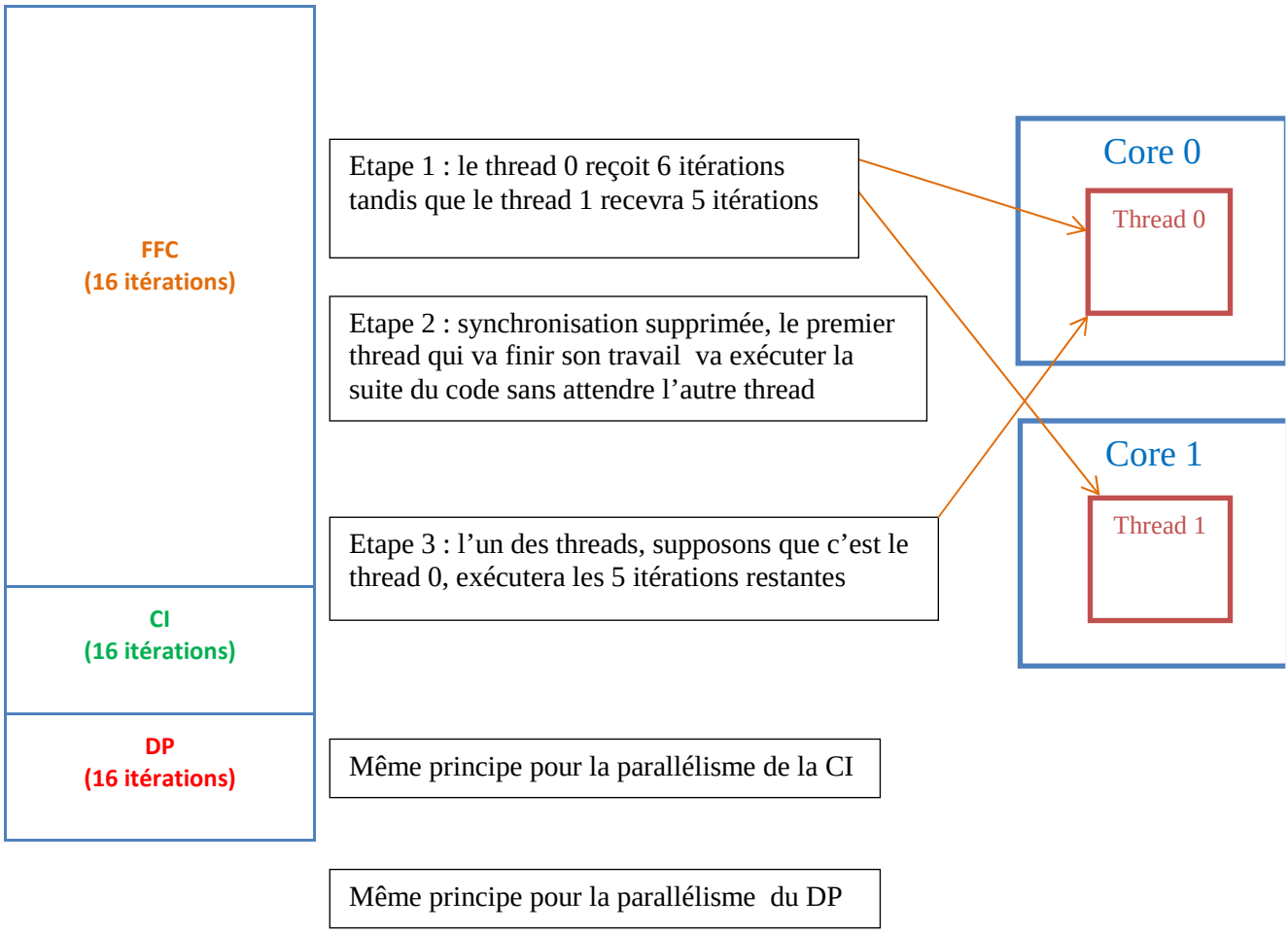


Figure 37 : Parallélisme sans synchronisation

Description détaillée :

Dans une région parallèle, on va ne construction SECTIONS.
Dans la construction FOR, on mettra que 11 itérations avec une répartition guidée, càd qu'un thread exécutera 6 itérations et l'autre exécutera 5 itérations. On supprime la barrière de synchronisation présente à la fin de la construction FOR.
Dans la construction SECTIONS, on mettra alors les 5 itérations qui restent.
On aura ainsi exécutés les 16 itérations qu'on avait dans le cas séquentiel.

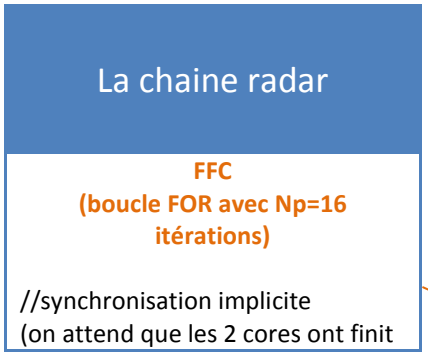
Directives OpenMP utilisées :

#pragma omp parallel : pour créer les trois régions parallèles, chacune correspond à un bloc de la chaine radar.
#pragma omp for schedule(guided,5) nowait: utilisée à l'intérieur de la région parallèle pour paralléliser une boucle for allant de 0 à 11. Càd qu'un thread exécutera 6 itérations alors que l'autre exécutera 5 itérations. La clause nowait permet de supprimer la barrière en fin de construction FOR.
#pragma omp for sections : sert à informer le compilateur de la présence d'une section du code à paralléliser par la suite
#pragma omp for section : sert à déclarer la section du code qui va être associé à un thread. Dans notre cas, on n'utilise qu'une seule section pour qu'elle soit exécutée avec un seul thread.

Sérialisation d'une section du code à l'intérieur d'une région parallèle

Dans ce cas d'implémentation, on testera la sérialisation d'une portion du code à l'intérieur de la région parallèle à l'aide de la construction SINGLE.
On utilisera ce cas à l'intérieur de la région parallèle correspondante à la FFC.
On parallélise statiquement la boucle for externe de chaque bloc de la chaine sur les 2 cores en utilisant deux threads.

Schéma de parallélisme :



Etape initiale : le thread 0 initialisera un tableau qui va contenir des informations nécessaires pour le calcul de la FFC. Une synchronisation sera faite à la fin de cette construction avant de commencer dans le calcul de la FFC

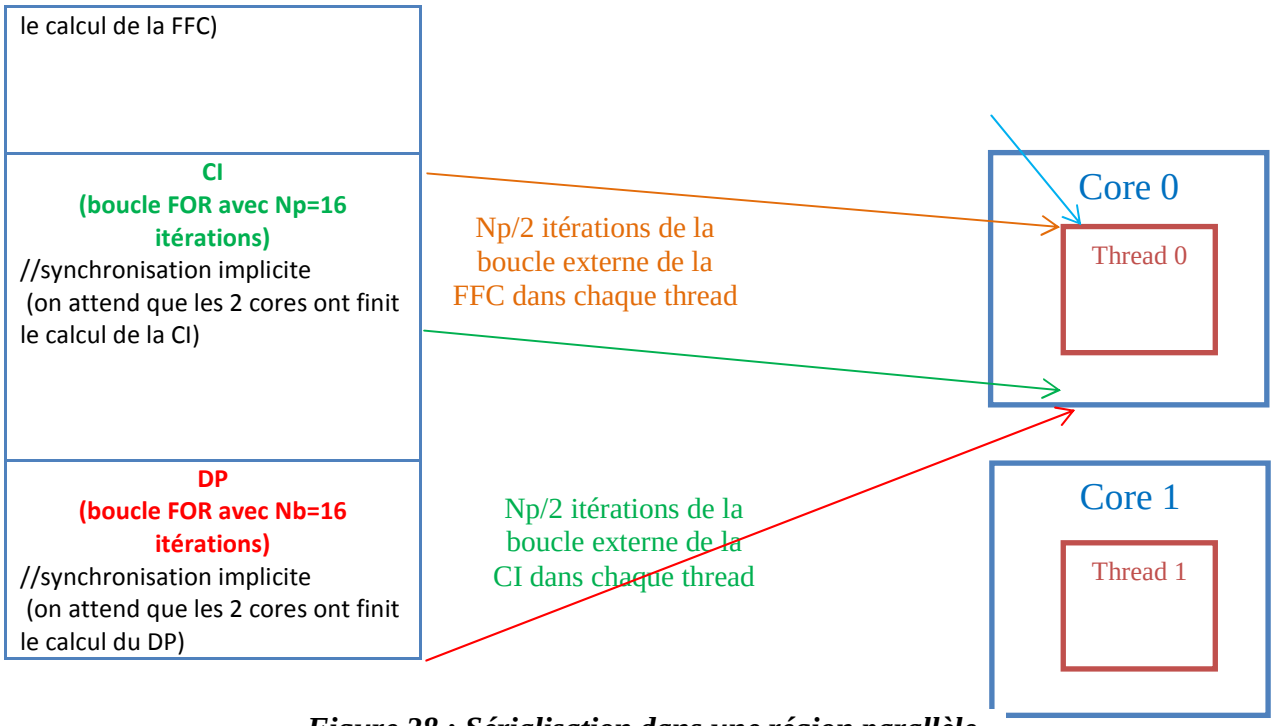


Figure 38 : Sérialisation dans une région parallèle

boucle externe du DP
dans chaque thread

Description détaillée :

A l'entrée de la région parallèle, le thread 0 initialisera un tableau qui contient les valeurs des variables qui parcourent les vecteurs d'entrée et de sortie de la FFC. Ces valeurs dépendent de la matrice calculée. On garde une synchronisation à la fin de ce bloc pour s'assurer que l'initialisation a été faite avant de commencer le calcul de la FFC.
Ensuite, on parallélise la boucle for externe de chaque bloc radar sur deux threads.
Dans le cas de la formation de faisceaux par le calcul et la compression d'impulsion, la boucle externe correspond au parcours des pulses (Np=16)
Dans le cas du filtrage doppler, la boucle externe correspond au parcours des faisceaux (Nb=16)
La répartition des itérations entre les deux threads se fait statiquement. Chaque thread va exécuter 8 itérations.
On garde la barrière de synchronisation présente à la fin de la construction FOR.

Directives OpenMP utilisées :

#pragma omp parallel : pour créer les trois régions parallèles, chacune correspond à un bloc de la chaine radar.
#pragma omp single : pour exécuter une section du code que par un seul thread. Cette construction contient une barrière implicite à la fin.
#pragma omp for schedule(static) : utilisée à l'intérieur de chaque région pour paralléliser une boucle for. On utilise une répartition statique des itérations de la boucle entre les deux threads.

Résultats

Ci-dessous, nous présenteronsles résultats fournis par les différentes implémentations présentées dans le paragraphe précédent:

Parallélisme appliqué sur des boucles for

	Référence en séquentielle (ms)	Parallélisme statique gain en %	Parallélisme dynamique gain en %	Parallélisme guidé gain en %
FFC	144,322	97,330	74,378	97,0440
CI	458,621	99,497	78,713	98,576
DP	44,281	91,312	79,851	96,125
Total=la chaine radar	647,224	98,405	77,778	98,059

Tableau 6 : Résultats du parallélisme des boucles

Parallélisme appliqué sur des sections du code

	Référence en séquentielle (ms)	Avec synchronisation gain en %	Sans synchronisation gain en %
FFC	144,322	71,053	78,323
CI	458,621	72,376	79,585

DP	44,281	69,914	77,308
Total=la chaine radar	647,224	71,905	79,141

Tableau 7 :Résultats du parallélisme des sections de code

Sérialisation d'une section du code à l'intérieur d'une région parallèle

	Référence en séquentielle (ms)	Gain en %
FFC	144,322	96,703
CI	458,621	99,883
DP	44,281	93,931
Total=la chaine radar	647,224	98,662

Tableau 8 : Résultats de la sérialisation d'une section dans une région parallèle

Analyse

Parallélisme appliqué sur des boucles for

Dans ce cas, on a créé trois régions parallèles chacune correspond à un bloc radar. Dans chaque région, on a parallélisé la boucle externe en répartionant les itérations de la boucle entre les deux threads. Cette répartition peut être réalisée de différentes manières :
Répartition statique : on a deux threads, alors ils se partagent la charge du travail et chacun exécutera la moitié des itérations. La répartition est totalement prédictible à l'avance et les deux threads supportent la même charge de travail.
Répartition dynamique : la charge de travail est non uniforme entre les threads. En fonction de la rapidité du thread, le gestionnaire de tâches lui affecte des paquets d'itérations à exécuter. Ce qui rend la répartition non prédictible à l'avance en plus de l'introduction d'un temps overhead.
Répartition guidée : Un cas particulier de la répartition dynamique. La répartition suit toujours le même principe sauf que dans ce cas, les paquets ont une taille décroissante. Ainsi, au départ on charge les threads avec des gros paquets d'itérations pour les maintenir en charge le plus longtemps possible, puis au fur et à mesure de l'avancement de l'exécution on leur attribue des paquets plus petits. On réduira ainsi le temps d'overhead qui posait problème dans le cas dynamique.

Parallélisme appliqué sur des sections du code

Dans le cas d'une implémentation avec synchronisation, on charge le premier thread avec 6 itérations à exécuter et l'autre thread avec 5 itérations, ensuite on synchronise. Quand les deux threads auront fini leur travail ils peuvent traverser la barrière et l'un d'eux continuera l'exécution du reste du code. Cette synchronisation ne fait que ralentir l'exécution du programme puisque le premier thread qui aura fini, surement celui qui n'a que 5 itérations à exécuter, peut continuer la suite du code sans avoir à attendre l'autre thread. Ceci se voit dans le gain fourni par chaque figure d'implémentation.

Sérialisation d'une section du code à l'intérieur d'une région parallèle

Dans la région parallèle, on crée une région séquentielle qui va être exécutée par un seul thread. On remarque que ceci ne réduit en rien les performances puisqu'on obtient des gains de parallélisme très proche de 100%.

Conclusion

A propos de la répartition la plus performante desitérations d'une boucle entre les threads, on recommande en premier lieu la répartition statique qui est prédictible et donne de bonnes performances. En deuxième lieu la répartition guidée à condition de bien tenir compte de la charge du travail et des threads disponibles. Il vaut mieux éviter la répartition dynamique qui introduit beaucoup d'overhead.

En fonction de l'application et du présent code, la synchronisation peut être gardée ou supprimée. Dans le cas où elle n'est pas nécessaire, comme le cas d'implémentation de la chaine radar présentée avant, il vaut mieux la supprimer pour accélérer l'exécution.

Si on juge nécessaire de sérialiser une section de code à l'intérieur d'une région parallèle, ceci peut se faire sans réduire les performances du code. En effet la construction SINGLE qui permet ceci ne ralentit pas l'exécution.

Conclusion générale

Notre mémoire est le fruit d'un travail de cinq mois, et sera l'un des axes principaux sur lequel se basera la définition de la machine de traitement parallèle pour la nouvelle génération des systèmes radar numériques de THALES AIR SYSTEMS.

L'objectif principal de mon stage est de supporter la thèse doctorale qui s'intitule « Architecture de Machines Parallèles pour le Traitement Intensif des futures générations de radar », dans le sens de l'étude du nouveau standard OpenMP. L'étude consiste en la compréhension du fonctionnement de ce dernier dans un premier lieu, ensuite décrire les composants qui pourront apporter des performances dans une architecture parallèle en accélérant au maximum le temps d'exécution. Et enfin implémenter certains modèles parallèles sur les blocs de la chaine radar qui seront utilisés par la suite dans la thèse.

Ce projet de fin d'études était pour nous l'occasion précieuse pour découvrir le monde des machines de traitement parallèle ainsi que les processeurs multi-coeurs, développer nos connaissances dans la programmation C, et surtout toucher au cœur de la problématique du traitement de signal intensif à savoir l'accélération du calcul. En termes de travail en équipe nous avons prouvé une grande homogénéité et un large partage de savoir. Le développement des connaissances techniques, esprit d'équipe et aspect ingénierie nous seraient utiles dans notre vie professionnelle.

Nos perspectives pour le projet, seront de tester OpenMP sur un processeur embarqué à mémoire partagécomme les DSP multi-coeurs de Texas Instruments et d'y implémenter la chaine de traitement radar pour l'adapter au temps réel. On testera ainsi l'API OpenMP sur une machine où le bas niveau est accessible, on propose de tester les performances d'un programme en activant et désactivant les caches L1 et L2 du processeur, en effectuant un transfert IDMA entre les caches pour voir la latence sans intervention du CPU. La construction de ce prototype sera la phase d'étude finale d'OpenMP qui permettra de lancer un verdict sur l'intégration d'OpenMP dans les nouveaux radars numériques de THALES AIR SYSTEMS.

Université Sidi Mohamed Ben Abdellah
Faculté des Sciences et Techniques Fès
Département Génie Electrique

Documents annexes

ANNEE UNIVERSITAIRE 2010-2011

Annexe 1 : [Compléments sur la présentation d'OpenMP]

Histoire OpenMP

Dans le début des années 90, les vendeurs de machines à mémoire partagée ont fourni des extensions de programmation Fortran:
L'utilisateur devrait écrire un programme Fortran en séquentiel avec les directives précisant les boucles qui devaient être parallélisées. Le compilateur est responsable de la parallélisation automatique de telles boucles sur les processeurs SMP. Première tentative de la norme a été le projet de norme ANSI X3H5 en 1994. Il n'a jamais été adopté, en grande partie à cause de baisse d'intérêt pour les SMP aussi les machines à mémoire distribuée sont devenues populaire.
La spécification standard OpenMP a commencé au printemps de 1997, quand de nouvelles architectures machine à mémoire partagée ont commencé à devenir courantes.[7]

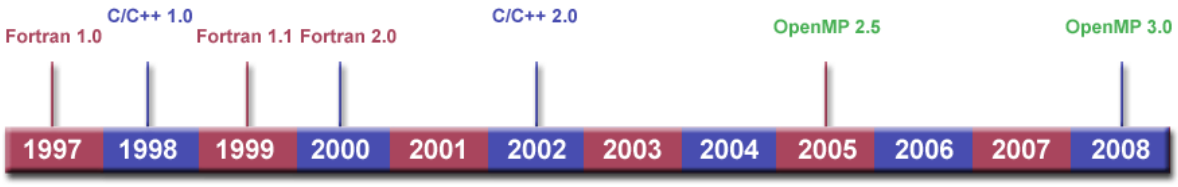


Figure 39 : Versions OpenMP

Le 7 Février 2011, la version 3.1 a été publiée pour les commentaires du public. La dernière spécification de la version 3.1 est prévue pour Juin 2011.

ARB OpenMP

OpenMP Architecture Review Board (ARB) est l'organisme qui possède la norme OpenMP, supervise la spécification OpenMP et produit et approuve de nouvelles versions de la spécification. L'ARB contribue à organiser et à financer des conférences, des ateliers et autres événements connexes, et encourage OpenMP. L'ARB est composé de membres permanents et auxiliaires. Les membres permanents sont les vendeurs qui ont un intérêt à long terme pour créer des produits pour OpenMP. Les membres auxiliaires sont normalement les organisations ayant un intérêt dans la norme, mais qui ne créent pas ou vend des produits OpenMP.[13]

Les membres permanents de l'ARB:

- ✓ AMD (Roy Ju)
- ✓ Cray (James Beyer)
- ✓ Fujitsu (Matthijs van Waveren)
- ✓ HP (Uriel Schafer)
- ✓ IBM (Kelvin Li)
- ✓ Intel (Sanjiv Shah)
- ✓ NEC (Kazuhiro Kusano)
- ✓ The Portland Group, Inc. (Michael Wolfe)
- ✓ Oracle Corporation (Nawal Copty)
- ✓ Microsoft (-)
- ✓ Texas Instruments (Andy Fritsch)
- ✓ CAPS-Entreprise (Francois Bodin)

Les membres auxiliaires de l'ARB:

- ✓ ANL (Kalyan Kumaran)
- ✓ ASC/LLNL (Bronis R. de Supinski)
- ✓ cOMPunity (Barbara Chapman)
- ✓ EPCC (Mark Bull)
- ✓ LANL (John Thorp)
- ✓ NASA (Henry Jin)
- ✓ RWTH Aachen University (Dieter an Mey)

Compilateurs OpenMP

Un nombre important de compilateurs ont intégré OpenMP API dans leurs environnements, ci-dessous la liste de ces compilateurs qui ont adopté la norme jusqu'à ce jour.[14]

Vendor/Source	Compiler
»GNU	GCC (4.3.2)
»IBM	XL C/C++ / Fortran
»Oracle	C/C++ / Fortran
»Intel	C/C++ / Fortran (10.1)
»Portland Group Compilers and Tool	C/C++ / Fortran
»Absoft Pro Fortran	Fortran
»Lahey/Fujitsu Fortran 95	C/C++ / Fortran
»PathScale	C/C++ / Fortran
»HP	C/C++ / Fortran
»MS	Visual Studio 2008-2010 C++
»Cray	Cray C/C++ and Fortran

Tableau 9 : Compilateurs OpenMP

Détails des directives et clauses de OpenMP

Construction d'une région parallèle

Au sein d'une région parallèle, tous les threads exécutent le même code.
Il existe une barrière implicite de synchronisation en fin de région parallèle.
Il est interdit d'effectuer des < branchements > (ex. GOTO, CYCLE, etc.) vers l'intérieur ou vers l'extérieur d'une région parallèle ou de toute autre construction OpenMP.[10]

Syntaxe : #pragma omp parallel [clause1] [clause2] ....

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
omp_set_num_threads(2); //specify 2 threads
#pragma omp parallel
{
printf("Hello world\n");
}
system("pause");
}
```

```
Hello world
Hello world
Appuyez sur une touche pour continuer...
```

Clause IF

La clause IF est supportée par une construction parallèle seulement, où elle est utilisée pour spécifier une exécution conditionnelle

Étant donné que la création et la terminaison d'une région parallèle, consomme inévitablement des ressources il est parfois nécessaire de tester s'il y'a suffisamment de travail dans la région pour justifier sa parallélisation. Le principal but de cette clause est de permettre un tel test. Si l'expression logique est évaluée à vrai, la région parallèle sera exécuté par une équipe de threads. Si elle renvoie false, la région est exécuté par un thread unique.

```
#pragma omp parallel if (n > 5)
private(TID) shared(n)
{
```

```
TID = omp_get_thread_num();
#pragma omp single
{
printf("Value of n = %d\n",n);
printf("Number of threads in parallel region: %d\n",
omp_get_num_threads());
}
printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --*/
```

Exécution avec n=7

```
Value of n = 7
Number of threads in parallel region: 4
Print statement executed by thread 3
Print statement executed by thread 1
Print statement executed by thread 2
Print statement executed by thread 0
Appuyez sur une touche pour continuer...
```

Exécution avec n=3

```
Value of n = 3
Number of threads in parallel region: 1
Print statement executed by thread 0
Appuyez sur une touche pour continuer...
```

Clause NUM_THREADS

La clause num_threads est supportée par une construction parallèle seulement, où elle est utilisée pour spécifier le nombre de threads qui doivent exécuter cette région parallèle

Dans l'exemple ci-dessous, on démontre que cette clause a plus de priorité que l'appel de la fonction omp_set_num_threads()

```
omp_set_num_threads(2);
#pragma omp parallel num_threads(8)
{
#pragma omp single
{
printf("Number of threads in parallel region: %d\n", omp_get_num_threads());
}
} /*-- End of parallel region --*/
```

```
Number of threads in parallel region: 8
Appuyez sur une touche pour continuer...
```

Variable partagée

Une variable partagée est définie dans un espace mémoire partagée, donc accessible par tous les threads (en lecture et écriture), sa valeur à l'entrée de la région est la même pour les threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int x=1;
omp_set_num_threads(2); // specify 2 threads
printf("Before parallel region ===>  x= %d\n\n",x);
#pragma omp parallel shared(x)
{
printf("thread %d ===>  x= %d\n", omp_get_thread_num(),x);
}
printf("\nAfter parallel region ===>  x= %d\n",x);
system("pause");
}
```

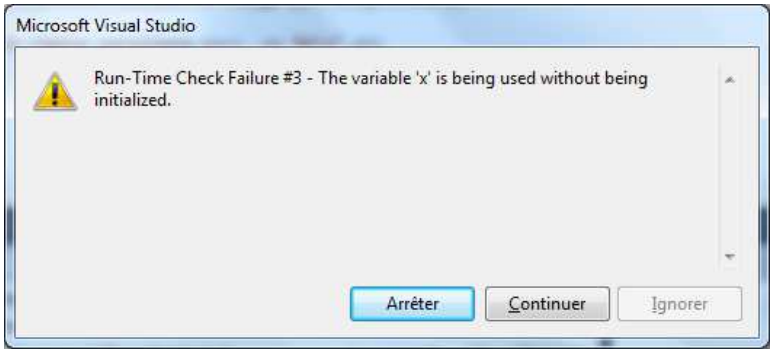
```
Before parallel region ===>  x= 1

thread 0 ===>  x= 1
thread 1 ===>  x= 1

After parallel region ===>  x= 1
Appuyez sur une touche pour continuer...
```

Variable privée

Une **variable privée** est définie dans la pile de chaque thread, elle n'est accessible qu'à l'intérieur du thread. Elle a une valeur indéterminée à l'entrée de la région parallèle. Les

modifications qu'elle va subir ne seront visible qu'à l'intérieur du thread; en effet, chaque thread dispose de sa propre copie de la variable. A la sortie de la région parallèle sa valeur est indéterminée.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int x=1;
omp_set_num_threads(2); // specify 2 threads
printf("Before parallel region ===>  x= %d\n\n",x);
#pragma omp parallel private(x)
{
printf("thread %d ===>  x= %d\n", omp_get_thread_num(),x);
}
printf("\n Before parallel region ===>  x= %d\n",x);
system("pause");
}
```



Variable firstprivate

La clause firstprivate: rend sa liste de variables privées et les initialise avec la valeur présente en mémoire partagée.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int x=1;
```

```
omp_set_num_threads(2); // specify 2 threads
printf("Before parallel region ===>  x= %d\n\n",x);
#pragma omp parallel firstprivate(x)
{
printf("thread %d ===>  x= %d\n", omp_get_thread_num(),x);
x=x+1;
}
printf("\nAfter parallel region ===>  x= %d\n",x);
system("pause");
}
```

```
Before parallel region ===>  x= 1

thread 0 ===>  x= 1
thread 1 ===>  x= 1

After parallel region ===>  x= 1
Appuyez sur une touche pour continuer...
```

Variable lastprivate

La clause lastprivate: rend sa liste de variables privées et enregistre la dernière modification dans la variable d'origine en mémoire partagée

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int i,n=2;
int x=1;
omp_set_num_threads(2); // specify 2 threads
printf("Before parallel region ===>  x= %d\n\n",x);
#pragma omp parallel for firstprivate(x) lastprivate(x)
for(i=0;i<n;i++)
{
printf("thread %d ===>  x= %d\n", omp_get_thread_num(),x);
x=x+1;
}
```

```
printf("\nAfter parallel region ===>  x= %d\n",x);

system("pause");

}
```

```
Before parallel region ===>  x= 1

thread 0 ===>  x= 1
thread 1 ===>  x= 1

After parallel region ===>  x= 2
Appuyez sur une touche pour continuer...
```

Partage de travail

À l'entrée d'une région parallèle les threads créés exécutent le même code. Il n'y a aucun intérêt à ceci.

Il faut créer des constructions pour pouvoir partager le travail entre les threads.

Pour se faire, OpenMP fournit deux constructions principales:

Paralléliser une boucle FOR à l'aide de: *#pragma omp FOR*

Paralléliser des portions du code à l'aide de: *#pragma omp SECTIONS*

Boucle parallèle

C'est un parallélisme par répartition des itérations d'une boucle.

La boucle parallélisée est celle qui suit immédiatement la directive for.

Les boucles < infinies > et do while ne sont pas parallélisables avec OpenMP.

Par défaut, une synchronisation globale est effectuée en fin de construction for.

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

int i,n=9;

omp_set_num_threads(4); //specify 4 threads

#pragma omp parallel

{

#pragma omp for

for (i=0; i<n; i++)
```

```
printf("Thread %d executes loop iteration %d\n", omp_get_thread_num() , i );
}
system("pause");
}
```

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 1 executes loop iteration 4
Appuyez sur une touche pour continuer...
```

Clause SCHEDULE - répartition STATIC

La clause SCHEDULE fixe la politique de distribution des itérations entre les threads

La répartition STATIC: les itérations sont réparties entre les threads en bloque de tailles égales à chunk en suivant le principe du tourniquet, si chunk n'est pas définit alors on va répartir des bloques contigus de tailles égales.

#pragma omp for schedule(static [,chunk])

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int i,n=9;
omp_set_num_threads(4); // specify 4 threads
#pragma omp parallel schedule(static,1)
{
#pragma omp for
for (i=0; i<n; i++)
printf("Thread %d executes loop iteration %d\n", omp_get_thread_num() , i );
}
system("pause");
}
```

```
Thread 1 executes loop iteration 1
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 4
Thread 0 executes loop iteration 8
Thread 1 executes loop iteration 5
Thread 2 executes loop iteration 2
Thread 2 executes loop iteration 6
Thread 3 executes loop iteration 3
Thread 3 executes loop iteration 7
Appuyez sur une touche pour continuer...
```

Clause SCHEDULE - répartition DYNAMIC

La répartition DYNAMIC: les itérations sont réparties entre les threads en bloque de tailles égales à chunk, lorsque un thread épuise sont bloque on lui affecte un autre bloque et ainsi de suite. Les bloques sont donc affectés dynamiquement. Si chunk n'est pas définit alors sa valeur est 1.

#pragma omp for schedule(dynamic [,chunk])

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int i,n=9;
omp_set_num_threads(4); // specify 4 threads
#pragma omp parallel schedule(dynamic,1)
{
#pragma omp for
for (i=0; i<n; i++)
printf("Thread %d executes loop iteration %d\n", omp_get_thread_num() , i );
} /*-- End of parallel region --*/
system("pause");
}
```

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 3
Thread 3 executes loop iteration 6
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 0 executes loop iteration 4
Thread 1 executes loop iteration 1
Thread 2 executes loop iteration 5
Appuyez sur une touche pour continuer...
```

Clause SCHEDULE - répartition GUIDED

La répartition GUIDED: les itérations sont divisées en paquets dont la taille décroit exponentiellement. Tous les paquets ont une taille supérieure ou égale a une valeur donnée (chunk). Si tôt qu'une tâche finit ses itérations, un autre paquet d'itérations lui est attribué. Si chunk n'est pas définit alors sa valeur est 1.

#pragma omp for schedule(guided [,chunk])

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
int i,n=9;
omp_set_num_threads(4); // specify 4 threads
#pragma omp parallel schedule(guided,3)
{
#pragma omp for
for (i=0; i<n; i++)
printf("Thread %d executes loop iteration %d\n", omp_get_thread_num() , i );
} /*-- End of parallel region --*/
system("pause");
}
```

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 0 executes loop iteration 6
Thread 0 executes loop iteration 7
Thread 0 executes loop iteration 8
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
Thread 1 executes loop iteration 5
Appuyez sur une touche pour continuer...
```

Clause SCHEDULE - répartition runtime et auto

runtime : la stratégie est fixée par la variable d'environnement OMP_SCHEDULE, dans ce cas la valeur de chunk ne doit pas être fixée.

#pragma omp for schedule(runtime)

auto : la stratégie à appliquer est choisie par le compilateur et/ou l'environnement d'exécution.

#pragma omp for schedule(auto)

Sections parallèles

Une section est une portion de code exécutée par une et une seule tâche.

Plusieurs portions de code peuvent être définies par l'utilisateur à l'aide de la directive SECTION au sein d'une construction SECTIONS.

Le but est de pouvoir répartir l'exécution de plusieurs portions de code indépendantes sur les différentes threads.

En fin de construction il existe une barrière de synchronisation implicite.

#pragma omp sections [clause ...]

{

#pragma omp section

structured_block

#pragma omp section

structured_block

#pragma omp section

structured_block

}

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
omp_set_num_threads(2); // specify 2 threads
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
printf("thread %d execute section A\n", omp_get_thread_num());
#pragma omp section
printf("thread %d execute section B\n\n", omp_get_thread_num());
}
} /*-- End of parallel region --*/
system("pause");
```

```
}
```

```
thread 0 execute section A
thread 1 execute section B

Appuyez sur une touche pour continuer...
```

Clause nowait

La clause nowait permet au programmeur d'affiner les performances d'un programme.

Quand on a introduit le partage du travail, nous avons mentionné qu'il y'a une barrière implicite à la fin de la construction. Cette clause permet d'éliminer cette barrière implicite. En d'autres termes, si cette clause est ajoutée, cette barrière sera supprimé.

Notez, cependant, que la barrière à la fin d'une région parallèle ne peut être supprimée.

L'utilisation est simple. Une fois un programme parallèle fonctionne correctement, on peut essayer d'identifier les endroits où une barrière n'est pas nécessaire et insérer la clause nowait.

L'exemple de code illustré ci-dessous montre son utilisation dans le code C. Quand un thread va atteindre la fin de la construction FOR, il va immédiatement continuer son travail sans attendre l'arrivée des autres threads.

Combinaison parallel/for et parallel/sections

Il est possible de combiner une directive parallel à soit sections soit for, cela s'avère utile si la région parallèle se résume à une boucle (for) ou à une distribution des tâches (sections).

```
#pragma omp parallel for
for(i=0;i<N;i++)
{}
```

Au lieu de

```
#pragma omp parallel
{
#pragma omp  for
for(i=0;i<N;i++)
{}
}
```

```
#pragma omp parallel sections
{}
```

Au lieu de

```
#pragma omp parallel
{
```

```
#pragma omp  sections
{}
}
```

Exécution exclusive

Il arrive que l'on souhaite exécuter séquentiellement certaines portions de code incluses dans une région parallèle.

Pour se faire, OpenMP propose deux directives SINGLE et MASTER.

Construction SINGLE

La construction SINGLE permet de faire exécuter une portion de code par un seul thread sans pouvoir indiquer lequel.

En général, c'est le thread qui arrive le premier sur la construction SINGLE.

Tous les threads n'exécutant pas la région SINGLE attendent, en fin de construction SINGLE, la terminaison de celui qui en a la charge.

#pragma omp single [clause1] ....

```
omp_set_num_threads(4);
#pragma omp parallel shared(a,b) private(i)
{
#pragma omp single
{
a = 10;
printf("Single construct executed by thread %d\n", omp_get_thread_num());
} /* A barrier is automatically inserted here */
#pragma omp for
for (i=0; i<n; i++)
b[i] = a;
} /*-- End of parallel region --*/
printf("After the parallel region:\n");
for (i=0; i<n; i++)
printf("b[%d] = %d\n",i,b[i]);
```

```
Single construct executed by thread 0
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
Appuyez sur une touche pour continuer... _
```

Clause COPYPRIVATE

La clause COPYPRIVATE permet de diffuser les valeurs acquises par un thread à toutes les instances des variables privées dans les autres threads. L'effet de cette clause est après le bloc construit par la directive SINGLE et avant que les threads ne traversent la barrière implicite à la fin de la région.

Construction MASTER

La construction MASTER permet de faire exécuter une portion de code par la tâche maitre seule.

Cette construction n'admet aucune clause.

Il n'existe aucune barrière de synchronisation ni en début (MASTER) ni en fin de construction.

#pragma omp master

```
omp_set_num_threads(4);
#pragma omp parallel shared(a,b) private(i)
{
#pragma omp master
{
a = 10;
printf("Master construct is executed by thread %d\n", omp_get_thread_num());
}
#pragma omp barrier
/* A barrier is automatically inserted here */
#pragma omp for
for (i=0; i<n; i++)
b[i] = a;
} /*-- End of parallel region --*/
printf("After the parallel region:\n");
for (i=0; i<n; i++)
```

```
printf("b[%d] = %d\n",i,b[i]);
```

```
Master construct is executed by thread 0
After the parallel region:
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
Appuyez sur une touche pour continuer...
```

Synchronisations

Ces directives permettent d'assurer une cohérence entre l'exécution des différents threads.

Barrière

La directive BARRIER synchronise l'ensemble des threads dans une région parallèle.

Chacun des threads attend que touts les autres soient arrivées à ce point de synchronisation pour reprendre, ensemble, l'exécution du programme.

#pragma omp barrier

```
omp_set_num_threads(4);

#pragma omp parallel private(TID)

{

TID = omp_get_thread_num();

printf("thread %d before barrier\n", TID);

#pragma omp barrier

printf("thread %d after barrier\n", TID);

} /*-- End of parallel region --*/
```

```
thread 0 before barrier
thread 3 before barrier
thread 2 before barrier
thread 1 before barrier
thread 1 after barrier
thread 3 after barrier
thread 2 after barrier
thread 0 after barrier
Appuyez sur une touche pour continuer...
```

Mise à jour atomique

La directive ATOMIC assure qu'une variable partagée est lue et modifiée en mémoire par un seul thread à la fois.

Son effet est local à l'instruction qui suit immédiatement la directive.

#pragma omp atomic

```
omp_set_num_threads(4);
```

```
int ic, i, n=5;
ic = 0;
#pragma omp parallel for shared(n,ic) private(i)
for (i=0;i<n;i++)
{
#pragma omp atomic
ic++;
}
printf("counter = %d\n", ic);
```

```
counter = 5
Appuyez sur une touche pour continuer...
```

Régions critiques

Une région critique peut être vue comme une généralisation de la directive ATOMIC.

Les threads exécutent cette région dans un ordre non déterministe mais une à la fois.

Une région critique est définie grâce à la directive CRITICAL et s'applique à une portion de code.

Les noms des sections critiques sont globales, ainsi deux sections portant le même nom sont considérées comme étant la même.

De même les sections sans nom sont considérées aussi comme étant une même section. Il est donc nécessaire de nommer les sections indépendantes.

#pragma omp critical [name]

```
omp_set_num_threads(4);
#pragma omp parallel private(TID)
{
TID = omp_get_thread_num();
#pragma omp critical (region_1)
{
printf("Thread %d executes critical region 1\n",TID);
}
#pragma omp critical (region_2)
{
printf("Thread %d executes critical region 2\n",TID);
```

```
}

} /*-- End of parallel region --*/
```

```
Thread 0 executes critical region 1
Thread 1 executes critical region 1
Thread 0 executes critical region 2
Thread 1 executes critical region 2
Thread 3 executes critical region 1
Thread 3 executes critical region 2
Thread 2 executes critical region 1
Thread 2 executes critical region 2
Appuyez sur une touche pour continuer...
```

Directive FLUSH

Nous avons vu que le modèle de mémoire OpenMP fait la distinction entre les données partagées, qui sont accessible et visible pour tous les threads, et les données privées, qui sont local à un thread.

Lorsque les données sont partagées, les choses sont plus complexes qu'il ne semble paraitre. En effet, la plupart des processeurs ont leurs propres mémoires locales à grande vitesse, les registres et la mémoire cache (voir Fig). Si un thread mit à jour des données partagées, les nouvelles valeurs sont d'abord enregistrées dans un registre, puis stockés dans le cache. Les mises à jour ne sont donc pas nécessairement immédiatement visible par les autres threads, car les autres threads des autres processeurs n'ont pas accès à cette mémoire. Sur une machine cache-cohérente, la modification de cache de l'un des processeurs est diffusé aux autres processeurs afin de les informer des changements, mais les détails de comment et quand ceci est effectué dépend de la plate-forme.

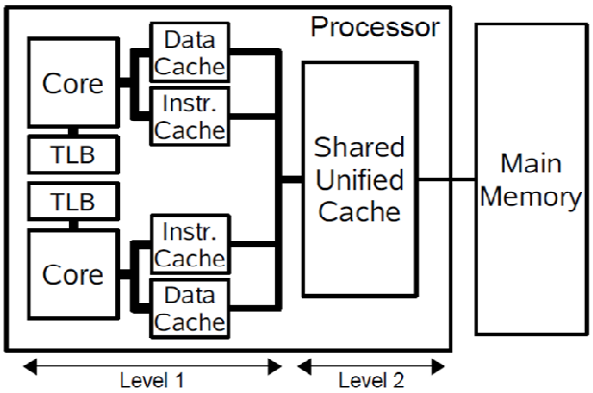


Figure 40 : Block diagram Core2Duo processor

OpenMP protège ses utilisateurs d'avoir besoin de savoir comment un ordinateur donné gère ce problème de cohérence des données. La norme OpenMP précise que toutes les modifications sont écrites dans la mémoire principale, à des points de synchronisation dans le programme et sont donc à la disposition de tous les threads. Entre ces points de synchronisation, les threads sont autorisés à avoir de nouvelles valeurs pour les variables partagées stockées dans leurs mémoire local plutôt que dans la mémoire globale partagée. En conséquence, chaque thread a sa propre vision temporaire des valeurs de données partagées. Cette approche, appelée relaxed consistency model facilite le travail du système pour offrir des bons performances.

Mais parfois les valeurs mises à jour des données partagées doivent devenir visibles aux autres threads entre-deux points de synchronisation. OpenMP fournit la directive FLUSH pour rendre cela possible.

L'objectif de cette directive est de rendre la vue temporaire d'un thread compatible avec les valeurs en mémoire.

#pragma omp flush([list-var])

Si la construction FLUSH est invoqué par un thread qui a mis à jour les variables, leurs nouvelles valeurs seront enregistrés en mémoire principale et ainsi ils sont accessibles à tous les autres threads

Si la construction est invoqué par un thread qui n'a pas mis à jour les données, il veillera à ce que toutes les copies locales des données seront remplacés par la dernière valeur de la mémoire principale.

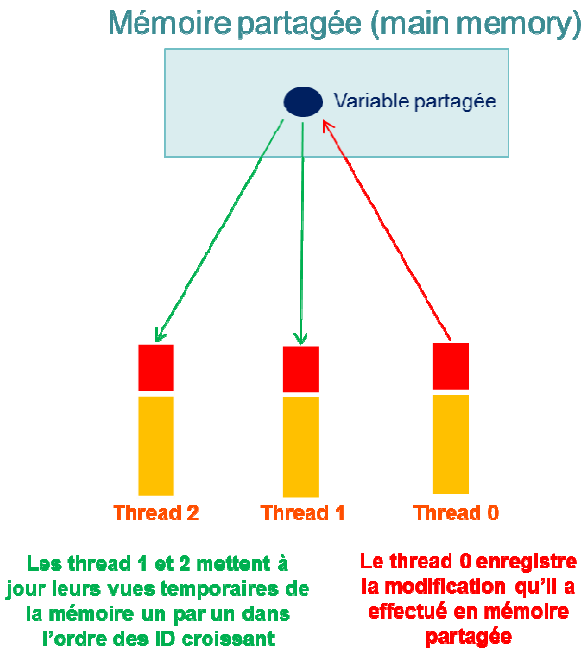


Figure 41 : Représentation du fonctionnement du Flush

Directive ordered

Elle permet, à l'intérieur d'une boucle parallélisée, d'exécuter une zone séquentiellement, c.à.d. dire thread par thread, dans l'ordre des indices croissant.

Cette directive ne peut être utilisé qu'à l'intérieur d'une directive for.

#pragma omp for ordered [clauses...]

(loop region)

#pragma omp ordered newline

structured_block

(endo of loop region)

```
omp_set_num_threads(4);
#pragma omp parallel for default(none) ordered schedule(runtime) private(i,TID) shared(n,a)
for (i=0; i<n; i++)
{
TID = omp_get_thread_num();
printf("Thread %d updates a[%d]\n",TID,i);
a[i] += i;
#pragma omp ordered
{
printf("Thread %d prints value of a[%d] = %d\n",TID,i,a[i]);
}
}  /*-- End of parallel for --*/
```

```
Thread 0 updates a[0]
Thread 0 prints value of a[0] = 0
Thread 0 updates a[1]
Thread 0 prints value of a[1] = 1
Thread 3 updates a[4]
Thread 1 updates a[2]
Thread 1 prints value of a[2] = 2
Thread 2 updates a[3]
Thread 2 prints value of a[3] = 3
Thread 3 prints value of a[4] = 4
Appuyez sur une touche pour continuer...
```

Directive THREADPRIVATE

Par défaut, un thread ne conserve pas ses copies des variables privées dans des régions parallèles différentes. Cependant, la directive threadprivate permet de dire à chaque thread de garder une même copie d'une variable privé entre les différentes régions. Ainsi, les modifications apportées sur une variable privée dans une région seront visibles dans la région suivante.

Cette directive se place immédiatement après la déclaration des variables dans le code source. Elle ne peut être utilisée que lorsque l'ajustement dynamique des threads est désactivée.

#pragma omp threadprivate(list-var)

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

```
int  a, b, i, tid;
float x;
#pragma omp threadprivate(a, x)
void main ()  {
/* Explicitly turn off dynamic threads */
  omp_set_dynamic(0);
 omp_set_num_threads(4);
printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
  {
tid = omp_get_thread_num();
  a = tid;
  b = tid;
  x = 1.1 * tid +1.0;
printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
}  /* end of parallel section */
printf("************************************\n");
printf("Master thread doing serial work here\n");
printf("************************************\n");
printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
  {
tid = omp_get_thread_num();
printf("Thread %d:   a,b,x= %d %d %f\n",tid,a,b,x);
}  /* end of parallel section */
system("pause");
}
```

```
1st Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 1:   a,b,x= 1 1 2.100000
Thread 3:   a,b,x= 3 3 4.300000
Thread 2:   a,b,x= 2 2 3.200000
************************************
Master thread doing serial work here
************************************
2nd Parallel Region:
Thread 0:   a,b,x= 0 0 1.000000
Thread 3:   a,b,x= 3 0 4.300000
Thread 2:   a,b,x= 2 0 3.200000
Thread 1:   a,b,x= 1 0 2.100000
Appuyez sur une touche pour continuer...
```

a et x ont conservé leurs valeurs entre la première région parallèle à la deuxième.

Clause REDUCTION

La clause REDUCTION rend les variables privés et applique sur elle une réduction pour déterminer la valeur des variables globales qui leur sont associées. operator doit être l'un des suivant : +, *, -, /, &, ^, |, &&, ||

reduction (operator: list-var)

```
const int n=5;
int i,a[n],b[n],result=0;
for (i=0; i<n; i++)
a[i] = i;
omp_set_num_threads(4);
#pragma omp parallel private(i)
{
#pragma omp for schedule(static) reduction(+:result)
for (i=0; i < n; i++)
result = result + a[i];
}
printf("result=%d\n\n",result);
```

```
result=10

Appuyez sur une touche pour continuer...
```

Les tasks

Une tâche a:

- ✓ Un code et des instructions à exécuter
- ✓ Un environnement de données (données propres à la tâche)
- ✓ Un thread qui va exécuter ce code et utiliser les données

OpenMP a toujours eu la notion de tâche, à la rencontre d'une région parallèle, des threads sont créés et chacun exécute une et une seule tâche. Chaque thread est assigné à une seule tâche et il est lié à elle.

Nouveauté dans OpenMP 3.0 :

- ✓ Un thread peut exécuter plus d'une tâche.
- ✓ Un thread peut switcher entre plusieurs tâches

Intérêt:

- ✓ Un algorithme peut être mieux implémenté en utilisant les task.
- ✓ l'imbrication de régions parallèles peuvent ralentir l'algorithme à cause des barrières implicites en fin de région.
- ✓ Les constructions de tâches résoudent ce problème en respectant la structure du « fork and join » de OpenMP

Construction TASK

Elle définit une tâche explicite, qui peut être soit exécutée par le thread rencontré, ou la suspendre pour une exécution ultérieur.

#pragma omp task [clause]…

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
omp_set_num_threads(4);   //specify 4 threads
#pragma omp parallel
{
        #pragma omp single
        {
        printf("Thread %d executes single construct\n",omp_get_thread_num());
        #pragma omp task
        printf("Thread %d executes the child task \n",omp_get_thread_num());
        }
}
system("pause");
}
```

```
Thread 0 executes single construct
Thread 0 executes the child task
Appuyez sur une touche pour continuer...
```

Directive TASKWAIT

La directive TASKWAIT spécifie une attente d'achèvement des tâches enfants générés depuis le début de la tâche en cours.

Une sorte de garantie que toutes les tâches ont été exécutés.[10]

#pragma omp taskwait

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int fib(int n) {
        int i, j;
        if (n<2)
        return n;
        else {
#pragma omp task shared(i)
                i=fib(n-1);
#pragma omp task shared(j)
                j=fib(n-2);
#pragma omp taskwait
                return i+j;
        }
}
void main() {
        int x,n=6;
        omp_set_num_threads(2); //définir 4 threads
#pragma omp parallel
        {
#pragma omp single
        x=fib(n);
        }
```

```
        printf("Fibonacci number is %d\n\n", x);
        system("pause");
}
```

```
Fibonacci number is 8

Appuyez sur une touche pour continuer...
```

Annexe 2 : [Compilation OpenMP]

La façon avec laquelle OpenMP est implémentée a une influence non négligeable sur les performances du programme. Nous donnons ici un bref aperçu du processus de traduction de ces directives.
Un compilateur d'OpenMP doit identifier les directives d'OpenMP et les routines de bibliothèque dans un programme et les traduire d'un code source en un code objet multithread. C'est à la charge du développeur de définir une stratégie de parallélisme et d'insérer les directives nécessaires dans le code source. C'est à la charge du compilateur et son environnement d'exécution d'appliquer cette stratégie.
Pour que le compilateur lise les directives OpenMP, il faut lui spécifier un flag (une option) au moment de la compilation. Sinon, il va les considérer comme des commentaires et générer un code objet séquentiel.[15]

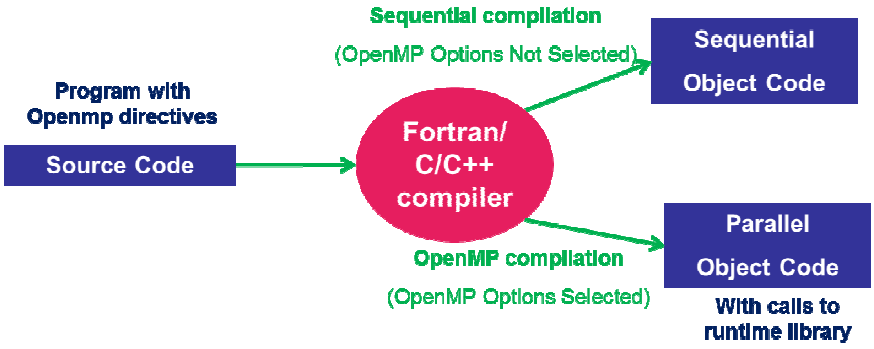


Figure 42 : Compilation OpenMP

Les bases de la compilation

Les compilateurs sont larges et comporte des packages complexes qui traduisent un code écrit avec un langage source donnée en un code objet correspondant à une machine cible.
Les détails du process sont propres à chaque compilateur, mais les stratégies sont similaires. La bibliothèque d'exécution et les détails de gestion de la mémoire aussi propres au compilateur.
De nombreux compilateurs sont capables de gérer de multiples langages sources et peuvent également générer du code pour plusieurs architectures cibles. Depuis que ce processus est devenu complexe, il est divisé en plusieurs phases.
Pendant la première phase, connue sous le nom de Front End, un programme source est lu, vérifié et converti en un format intermédiaire appropriée pour un traitement ultérieur. Le format est spécifique à un compilateur donné.
Pendant le reste de la compilation, les phases Middle End et Back End, le code intermédiaire est successivement modifié et simplifié jusqu'à ce qu'il soit converti en code machine pour la plate-forme cible.
Par la suite, on présentera les ajouts dans un compilateur OpenMP par rapport à un compilateur ordinaire.[15]

Les phases de la compilation

Front End

- ✓ Lire le language de base (Fortran, C ou C ++)
- ✓ Lire les directives OpenMP, analyse lexicale et syntaxique (directives incorrecte, imbrication incorrecte…)
- ✓ Créer une représentation intermédiaire avec des notations openmp pour plus de traitement par la suite [9]

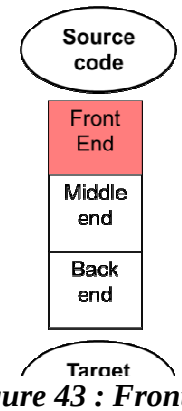


Figure 43 : Front End

Middle End

- ✓ Preprocess des constructions OpenMP
- ✓ Les barrières implicites deviennent explicites
- ✓ Plus de vérification d'exactitude
- ✓ Appliquer des optimisations
- ✓ Fusionner les régions parallèles adjacentes
- ✓ Fusionner les barrières adjacentes[9]

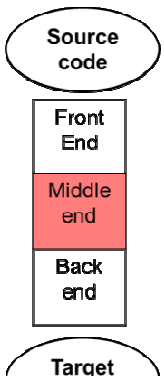


Figure 44 : Middle End

Back End

- ✓ Remplacer certaines constructions OpenMP par des appels à la bibliothèque d'exécution (exp: barrier,atomic,flush…)
- ✓ Implémenter les constructions parallèles en créant des tâches séparés qui vont être affectés aux threads
- ✓ Aussi, implémenter les attributs des données (privé ou partagé) et faire les initialisations
- ✓ Les constructions OpenMP sont traduites en un code multithread
- ✓ Enfin le code est assemblé en un code assembleur puis transformée en un fichier objet pour l'édition des liens avec la bibliothèque d'exécution. [9]

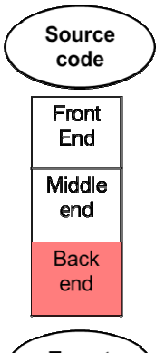


Figure 45 : Back End

Exemple d'un compilateur OpenMP

OpenUH est un compilateur de OpenMP open source supportant comme langages sources le C, C++ et le Fortran 77/90/95 et comme architectures cibles IA-64, x86, x86_64 Linux.
Le compilateur reconnait les directives OpenMP dans le Front End.[15]

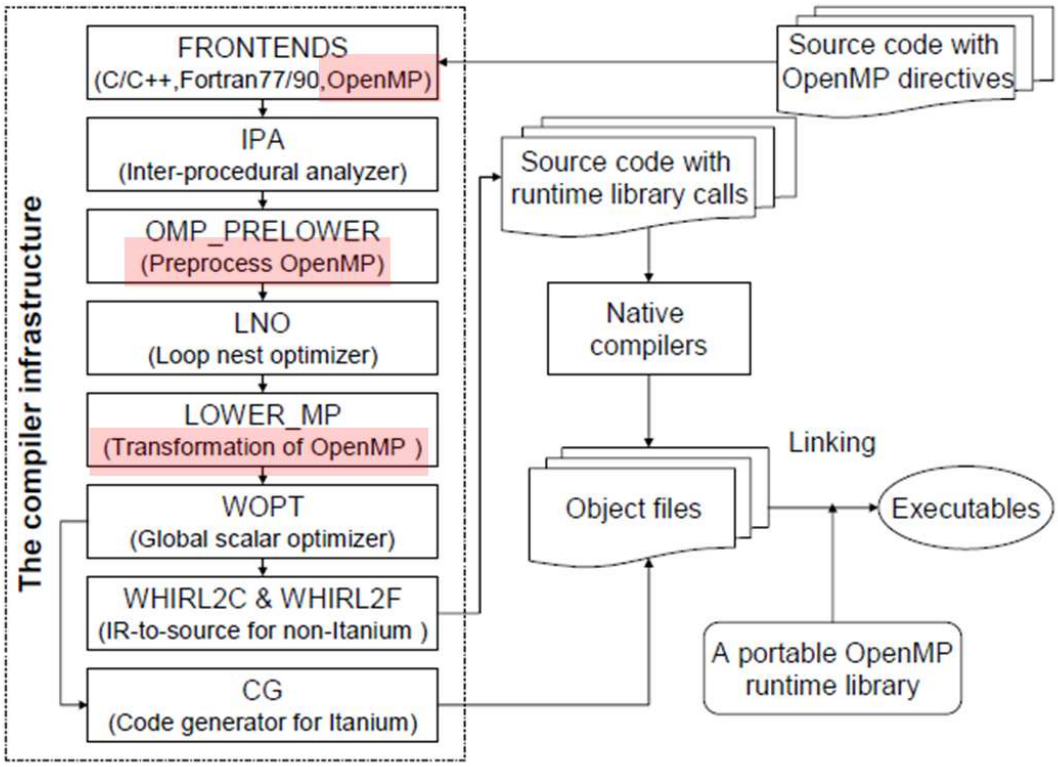


Figure 46 : Structure du compilateur OpenUH

Compilation conditionnelle

Une des fonctionnalités puissantes d'OpenMP est que l'on peut écrire un programme parallèle, tout en préservant le code source (original) séquentiel. Si l'on ne compile pas en utilisant l'option OpenMP adéquate, ou on utilise un compilateur qui ne prend pas en charge OpenMP, les directives sont tout simplement ignorées, et un exécutable séquentiel est généré.

Toutefois, OpenMP prévoit également l'exécution fonctions qui retournent des informations de l'environnement d'exécution. Pour veiller à ce que le programme sera encore compiler et exécuter correctement en mode séquentiel en leur présence, une attention particulière doit être prise lors de leur utilisation.

Par exemple, disons que l'on souhaite utiliser la omp_get_thread_num() qui renvoie l'ID du thread. Si l'application est compilée sans traduction OpenMP, le résultat sera une référence non résolue au moment de la liaison.

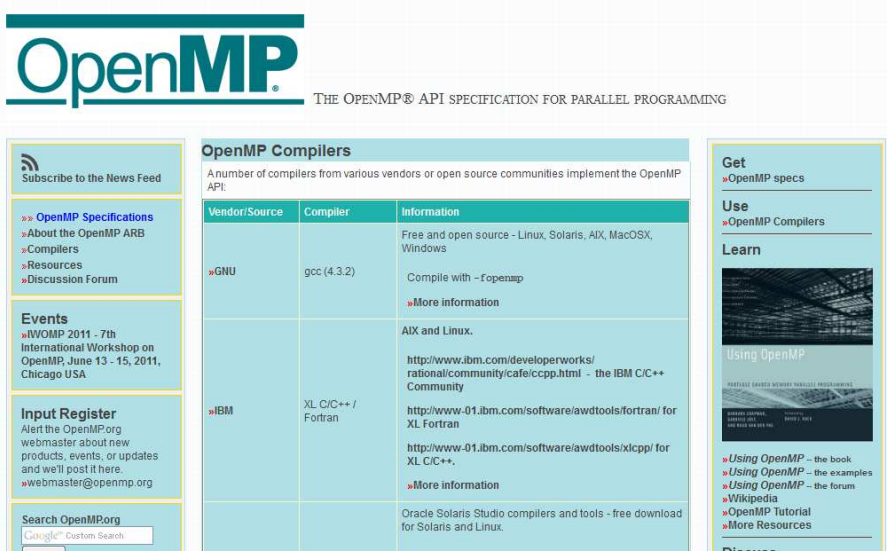
Plus précisément, les fonctions d'exécution OpenMP en C / C + + et Fortran peut être placé sous le contrôle d'un # ifdef _OPENMP, de sorte qu'ils ne seront traduits si la compilation OpenMP a été invoquée.

```
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
```

Annexe 3 : [Etapes d'exécutiond'un programme avec OpenMP]

Etape 1 :

Choisir un compilateur qui prend en charge la spécification OpenMP. Le choix est effectué aussi en fonction des besoins, la plateforme cible et le langage source.
On peut voir les compilateurs qui supportent la spécification sur ce lien : http://openmp.org/wp/openmp-compilers/



Etape 2 :

Ecrire le programme souce en C,C++ ou Fortran en établissant une stratégie de parallélisation avec l'insertion des directives de compilation et des fonctions de OpenMP.

Etape 3 :

Si on utilise des fonctions de OpenMP, il faut inclure le fichier omp.h dans le code source (#include <omp.h> en C/C++) *(en Fortran !$USE OMP_LIB)*
Ce fichier contient les prototypes des fonctions OpenMP.

Etape 4 :

Spécifier le nombre de threads à utiliser pour le parallélisme de l'application. On peut le faire de 3 façons différentes :

- ✓ En utilisant une fonction de OpenMP : omp_set_num_threads(nbr_threads) ;
- ✓ En utilisant une clause de OpenMP qui s'intègre avec la directive de création d'une région parallèle : #pragma omp parallel num_threads(nbr_threads){}
- ✓ En utilisant une variable d'environnement et ainsi le nombre de threads sera spécifié qu'au moment de l'exécution : SET OMP_NUM_THREADS=nbr_threads

Etape 5 :

Au moment de la compilation, spécifier un flag (option) pour que le compilateur tienne compte de la présence des directives OpenMP.
Ensuite un code objet multithread est généré par le compilateur. Ce code objet va être linkée avec la bibliothèque d'exécution (runtime library) de OpenMP pour finalement créer un exécutable.

Exemple simple en C:

```
#include<stdlib.h>
```

```
#include<stdlio.h>
#include<omp.h> //fichier d'inclusion qui contient les prototypes des fonctions d'OpenMP
int main()
{
omp_set_num_threads(2); // définir 2 threads
#pragma omp parallel // directive openmp pour créer une région parallèle
{
printf("Hello World\n");
}
return 0;
}
```

Exécution:

```
Hello World
Hello World
```

Annexe 4 : [Les coûts du parallélisme]

La programmation parallèle, malgré les promesses qu'elle apporte ne doit pas faire oublier qu'elle demande au programmeur une nouvelle façon d'écrire son code. Parmi les actions à effectuer, il est nécessaire d'étudier la qualité de la parallélisation.

Idéalement, lorsqu'un code est destiné à être exécuté sur un ordinateur multi-cœur, il devrait avoir une accélération linéaire, c'est-à-dire qu'en doublant le nombre de cœurs le temps d'exécution est divisé par deux. Pour un code de quelques lignes ça peut être le cas, mais pour une vraie application c'est rarement le cas. Pourquoi n'atteint-on jamais ces conditions idéales? Comment prévoir l'évolution du code si on augmente le nombre de cœurs? Est-il vraiment nécessaire de paralléliser? La faible accélération de l'exécution d'un programme vient-elle de la quantité de messages échangés, ou bien de la partie de l'algorithme non parallélisée ? C'est à ces questions que ce chapitre essayera de donner une première réponse en fournissant quelques outils mathématiques qui permettent de déterminer la qualité du code parallélisé.

Un processeur avec un seul coeur, qui doit effectuer une grosse tâche prennant une heure, ne mettrait plus qu'une demi-heure avec deux coeurs, quinze minutes avec quatre coeurs. Mais il arrivera inévitablement qu'à partir d'un certain nombre de coeurs les communications, les synchronisations, etc annulent les gains du parallélisme.

A un moment donné toute la meilleure organisation possible ne parviendrait à enrayer le problème suivant :

On passe plus de temps à communiquer, à s'informer les uns les autres qu'à travailler sur le problème.

De plus comment savoir si le ralentissement des performances provient de la partie séquentielle qui n'est donc pas parallélisée et qui est donc indépendante du nombre de coeurs, ou bien de la partie parallèle et de ses trop nombreux échanges qui ralentissent l'exécution?

Deux premières notions nous seront utiles, **l'accéleration** et **l'efficacité**.[16]

Accélération

L'accélération (Speed Up) constitue le rapport entre le temps séquentiel optimal par le temps parallèle.

$$\text{Accélération} = \frac{\text{Temps séquentiel}}{\text{Temps parrallèle}}$$

On comprend tout de suite que plus l'accélération s'approche de 1 et plus l'accélération due à la parallèlisation est nulle. Inversement, plus l'accélération s'approche du nombre de coeurs présents et plus l'exécution parallèle est intéressante car cela signifie que le travail a été correctement réparti entre les processeurs sans aucun coût supplémentaire (échanges de messages, synchronisation …).

Efficacité

L'efficacité d'une parallélisation en fonction d'un nombre de coeurs « P » s'obtient ensuite :

$$E = \frac{\text{Temps séquentiel}}{\text{Temps parrallèle} * P}$$

Plus l'efficacité E est proche de '1' et plus l'accélération est maximale.
1-E désigne le temps que l'algorithme a perdu en communications, synchronisations, etc.

Loi d'Amdahl

Une des premières lois permettant de mesurer le gain d'un ordinateur en modifiant le nombre de processeurs provient de Gene Amdahl.

En utilisant la formule d'accélération vue précédemment, Amdahl s'est contenté de réaliser une règle de trois ou Ψ constitue l'accélération :

Avec f : $\psi \leq \frac{1}{f + (1-f)/p}$ $\qquad f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)}$

σ(n):Temps consacré à la partie intrinsèquement séquentielle (non parallélisable)
φ(n): Efficacité obtenue en résolvant un problème de taille n avec p processeurs.

Dans les conditions idéales, les résultats de cette accélération s'approchent du nombre de cœurs présents, mais généralement ce n'est pas le cas car il faudrait que « f » soit nul. De plus, une des principales critiques de la loi d'Amdahl vient du fait qu'elle ne tient pas en compte de la complexité des communications, car généralement les résultats sont encore pires.

Loi de Gustafson-Barsis

La loi d'Amdahl ne tient pas compte du fait que la mise en place du parallélisme, les échanges de messages peuvent aussi influencer sur la durée d'un programme. Certains codes sont entièrement parallélisés, ce qui veut dire que 'f' est nul, or cela aurait pour signification que le programme suit la courbe de l'accélération idéale (elle est égale au nombre de coeurs).
La loi de Gustafson-Barsis vient combler en partie cette lacune, elle énonce que :

$$\psi \leq p + (1-p)s$$

Avec s la part d'exécution de la partie séquentielle : $$s = \frac{\sigma(n)}{\sigma(n) + \frac{\varphi(n)}{p}}$$

Métrique de Karp-Flatt

La métrique de Karp-Flatt est un instrument de mesure de la parallélisation dans un système multi-coeur. Elle a été proposée par Alan H. Karp et Horace P. Flatt en 1990. Cette métrique prend en compte les lois d'Amdahl et de Gustafson mais possède deux avantages sur eux :

- Elle prend en compte les coûts de la parallélisation
- Elle permet de détecter d'autres pertes ou inefficacités qu'on ne verrait pas avec les deux premières lois.

Par exemple, si on possède 19 tâches à exécuter sur six processeurs. Chaque tâche prenant une unité de temps complète. Le premier processeur aura 4 tâches à calculer les cinq autres en auront que 3 (1x4 + 3x5 = 19). Donc le temps d'exécution final dépendra uniquement du premier processeur qui a une tâche de plus. Le temps d'exécution est alors de 4 unités au lieu des 19/6 = 3.1 théoriques. Or ce constat se verrait immédiatement avec la métrique de Karp-Flatt.

Elle s'énonce de la manière suivante : $$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Avec comme accélération Ψ et p le nombre de processeurs où p>1

[Références]

[1] http://fr.wikipedia.org/wiki/Taxinomie_de_Flynn

[2] https://computing.llnl.gov/tutorials/parallel_comp/

[3] http://fr.wikipedia.org/wiki/Processus_%28informatique%29

[4] http://fr.wikipedia.org/wiki/Thread_%28informatique%29

[5] http://software.intel.com/fr-fr/articles/tnouidjem/

[6] Chapitre 2 du livre « Using OpenMP Portable Shared Memory Parallel Programming » Barbara Chapman, Gabriele Jost, Ruud van der Pas - The MIT Press 2008

[7] https://computing.llnl.gov/tutorials/openMP/#ProgrammingModel

[8] Cours OpenMP Ecole IN2P3 2010 Françoise Roch

[9] Slides « How OpenMP is Compiled » Barbara Chapman, Lei Huang University of Houston

[10] Specifications OpenMP Version 3.0 disponible sur http://openmp.org/wp/openmp-specifications/

[11] Rapport de stage de fin d'étude - David Combet - THALES -2010

[12] Rapport de stage de fin d'études - Julien Semplici- THALES -2010

[13] http://openmp.org/wp/about-openmp/

[14] http://openmp.org/wp/openmp-compilers/

[15] Chapitre 8 du livre « Using OpenMP Portable Shared Memory Parallel Programming

» Barbara Chapman, Gabriele Jost, Ruud van der Pas - The MIT Press 2008

[16] http://software.intel.com/fr-fr/articles/nouidjem-tareg/

Liste des figures

Liste des tables

Liste des acronymes

API	Application Programming Interface
ARB	Architecture Review Board
BF	Beam Forming
CD	Case Distance
CFAR	Constant False Alarm Rate
CPU	Central Processor Unit
DF	Doppler Filtring
DSP	Digital Signal Processor
FFC	Formation de Faisceaux par le Calcul
FLOPS	FLoating point Operations Per Second
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
NUMA	Non-Uniform Memory Access
OpenMP	Open Multi Processing
OS	Operating System
PC	Pulse Compression
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Symmetric Multi Processors
TI	Texas Instruments
UMA	Uniform Memory Access
VS	Visual studio

Université Sidi Mohamed Ben Abdellah
Faculté des Sciences et Techniques Fès
Département Génie Electrique

[Résumé]

Ce rapport porte sur «l'étudeet l'évaluation des performances d'OpenMP sur des processeurs multi-coeuravec application sur la chaine Radar».

Dans une première partie, on a étudié les performances des différents modules proposés par l'API de codage parallèle OpenMP en illustrant par des programmes exemples. On a utilisé des cas d'utilisation « use case » de traitement du signal pour une étude plus intéressante pour THALES.

Ensuite, on a appliqué quelques modules d'OpenMP sur trois blocs de la chaine radar afin de les paralléliser et ainsi accélérer le temps d'exécution au maximum. Enfin, on a réalisé quelques modèles de parallélismesur la chaine radar en analysant les performances apportés par chacune,pour en conclure des recommandations d'implémentation.

Ce stage a été concluant et nous a poussé à intégrer le monde des processeurs multi-coeur et les systèmes embarqués de façon générale.

[Abstract]

This report focuses on "the study and evaluation of OpenMP's performance on multi-core processors with application on the radar chain."

In the first part, we studied the performance of different modules offered by the OpenMP API for parallel programmingand we clarified by example programs. We used some usecase for signal processing, for a more interesting for THALES.

Then we applied some OpenMP's componentson three blocks of the radar chain in order to parallelize and thereby accelerate the execution time. Finally, wehave made some models of parallelism for the radar chain by analyzing performances provided by each, to conclude some recommendations for animplementation.

The internship was successful and led us to join the world of multi-core processors and embedded systems in general.

ANNEE UNIVERSITAIRE 2010-2011