

I.	Architecture parallèle	4
1.	Principe du parallélisme	4
2.	Applications du parallélisme	4
3.	Les différents types de parallélisme	4
a)	La taxonomie de Flynn	4
b)	Classification selon RAINA	6
4.	Calcul de la complexité parallèle	8
5.	Modèles de calcul parallèle	9
6.	Quelques modèles de calcul parallèle	10
a)	Le modèle PRAM (Parallel Random Access Memory)	10
b)	Le Modèle systolique	10
c)	Modèle grille à deux dimensions et modèle hypercube	11
d)	L'hypercube	12
II.	OPENMP	13
1.	Définition et Principe de fonctionnement	13
a)	Définition	13
b)	Principe Fonctionnement	14
2.	Structure d'un programme OpenMP	15
3.	Installation d'OpenMP	16
a)	Installation d'OpenMP sur windows	16
b)	Installation d'OpenMp sur linux	16
(1)	Compilation (Compilation avec GNU)	17
(2)	Exécution	17
4.	Variables avec OpenMP	17
a)	Portée des variables	17
b)	Statuts d'une variable	17
c)	Clause de la directive parallèle	18
d)	Étendue d'une région parallèle	20
e)	Compléments	20
5.	Partage du travail	21
a)	Les directives du partage du travail	22
(1)	Clause schedule	23
(2)	La clause nowait	24
(3)	La clause ordered	24
b)	Les directives de synchronisation	26
(1)	La directive BARRIER	26
III.	Avantages, inconvénients et limite de la programmation en OpenMP	27
1.	Avantages	27
2.	Inconvénients et limite de la programmation en OpenMP	27

Figure 1: Classe SISD -----	5
Figure 2 : Classe SIMD -----	5
Figure 3: Classe MISD -----	5
Figure 4: Classe MIMD -----	6
Figure 5: Récapitulatif de la taxonomie de Flynn -----	6
Figure 6: Classification MIMD de Raina -----	7
Figure 7: Modèle PRAM-----	10
Figure 8: Grille a deux dimensions de taille 16-----	12
Figure 9: Schéma illustratif régions parallèle -----	14
Figure 11: Utilisation de la clause private (nthreads) -----	18
Figure 10: Résultat du programme de la figure 10 -----	18

INTRODUCTION

En informatique le **parallélisme** consiste à mettre en œuvre des architectures d'électronique numérique permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible. 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 connaît des limites. La création de processeurs multi-cœurs, traitant plusieurs instructions en même temps au sein du même composant, résout ce dilemme pour les machines de bureau depuis le milieu des années 2000. Et il existe aussi des moyens de faire des programmations en parallèle à l'aide des api tels que OPENMP qui est celui qui nous intéresse.

OpenMP (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur les ordinateurs à mémoire partagée. C'est aussi un ensemble de directives de compilation et de bibliothèques de fonctions qui facilite la création des programmes multi-threads.

I. Architecture parallèle

1. Principe du parallélisme

En informatique, le **parallélisme** consiste à mettre en œuvre des architectures d'électronique numérique permettant de traiter des informations de manière simultanée, ainsi que les algorithmes spécialisés pour celles-ci. Ces techniques ont pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible. 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 connaît des limites. La création de processeurs multi-cœurs, traitant plusieurs instructions en même temps au sein du même composant, résout ce dilemme pour les machines de bureau depuis le milieu des années 2000.

Les premiers ordinateurs étaient séquentiels, exécutant les instructions l'une après l'autre. Le parallélisme se manifeste actuellement de plusieurs manières : en juxtaposant plusieurs processeurs séquentiels ou en exécutant simultanément des instructions indépendantes.

2. Applications du parallélisme

Certains types de calculs se prêtent particulièrement bien à la parallélisation : la dynamique des fluides, les prédictions météorologiques, la modélisation et simulation de problèmes de dimensions plus grandes, le traitement de l'information et l'exploitation des données, le décryptage de messages, la recherche de mots de passe, le traitement d'images ou la fabrication d'images de synthèse, tels que le lancer de rayon, l'intelligence artificielle et la fabrication automatisée. Tout d'abord, c'est dans le domaine des supercalculateurs que le parallélisme a été utilisé, à des fins scientifiques.

3. Les différents types de parallélisme

a) La taxonomie de Flynn

La taxonomie de Flynn est une classification des architectures d'ordinateur, proposée par Michael Flynn en 1966. Les quatre catégories définies par Flynn sont classées selon le type d'organisation du flux de données et du flux d'instructions.

Les différents types de classes

-SISD (Single Instruction Single Data) : Il s'agit d'un ordinateur séquentiel qui n'exploite aucun parallélisme, tant au niveau des instructions qu'au niveau de la mémoire. Cette catégorie correspond à l'architecture de Von Neumann. C'est illustrer par le schéma suivant :

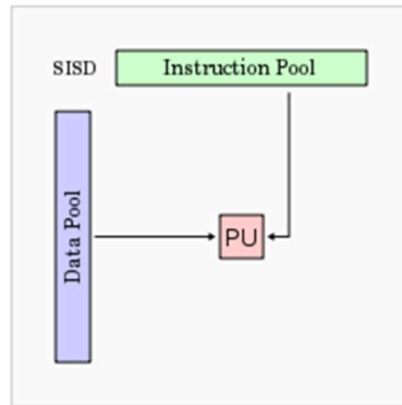


Figure 1: Classe SISD

-SIMD (Single Instructions Multiple Data) : C'est une architecture capable d'exécuter plusieurs données à la fois. Avec cette solution, les processeurs exécutent un seul et unique programme ou une suite d'instruction, mais chaque processeur travail sur une donnée différente.

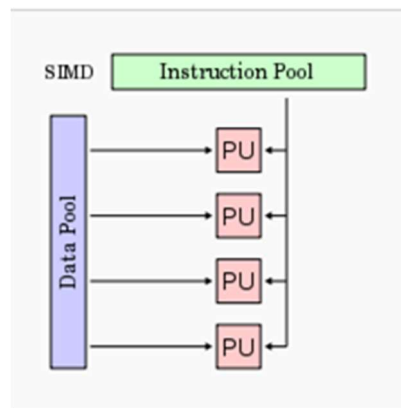


Figure 2 : Classe SIMD

-MISD (Multiple Instructions Single Data) : Exécutent des instructions différentes en parallèle sur une donnée identique. Cette catégorie d'architectures est vraiment très rare. On peut citer comme exemples les architectures systoliques et cellulaire.

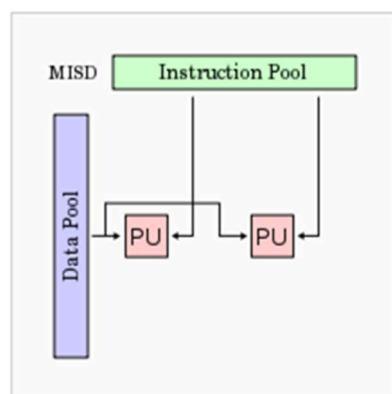


Figure 3: Classe MISD

-MIMD (Multiple Instructions Multiple Data) : Architectures capables d'exécuter une instruction différente sur des données différentes. Les processeurs multi cœurs et multiprocesseurs font partie de la catégorie MIMD. Elle est découpée en deux sous catégories :

*Le Single Program Multiple Data ou SPMD consiste à exécuter un seul programme sur plusieurs données à la fois. La différence avec le SIMD est que le SPMD peut exécuter des instructions différentes d'un même programme sur des données.

* Le Multiple Program Multiple Data ou MPMD qui consiste à exécuter des programmes en parallèle sur des données différentes. On peut ainsi découper un programme en sous-programme indépendant exécutable en parallèle ou exécuter plusieurs copie d'un programme. Dans les deux cas chaque copie ou morceau de programme est appelé thread.

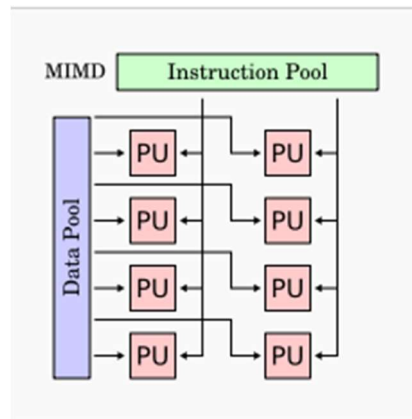


Figure 4: Classe MIMD

\	Données traitées en série	Données traitées en parallèle
Instructions traitées en série	SISD	SIMD
Instructions traitées en parallèle	MISD	MIMD

Figure 5: Récapitulatif de la taxonomie de Flynn

b) Classification selon RAINA

La classification de RAINA permet de prendre en compte de manière fine les architectures mémoires selon deux critères :

L'organisation de l'espace d'adressage

-Machine à mémoire partagée ou SASM (Single Adress Space, Shared Memory) :

Les processus ont accès à la mémoire comme un espace d'adressage global. Tout changement de case mémoire est vue par les autres processeurs.

La communication et la synchronisation inter-processeurs sont effectuées via la mémoire global (variables partagées).

-Machine à mémoire distribuées ou DADM (distributed Adress Space, Distributed Memory):

Chaque processeur a sa propre mémoire et son propre système d'exploitation. L'échange de donnée entre processeurs se fait par message.

-Architecture hybride ou SADM (Single Adress Space, Distributed Memory) : C'est l'architecture utilisée par les superordinateurs. Elle combine les caractéristiques des deux classes précédentes. La mémoire est physiquement distribuée sur les processeurs, cependant l'utilisateur a l'impression d'utiliser un seul espace d'adressage. Les systèmes hybrides possèdent l'avantage d'être extensible, performants et à faible coût.

Types d'accès mémoire mis en œuvre

-NORMA (No Remote Memory Access) : Il n'y a pas de moyen d'accès aux données distantes. La communication inter-processeurs se fait par passage de messages.

-UMA (Unirform Memory Access) : Les processeurs ont un accès symétrique la mémoire. Et les coûts sont identiques.

-NUMA (Non-Unirform Memory Access) : Les performances d'accès dépendent de la localisation des données. L'accès en temps constant à la mémoire étant une contrainte technologique difficile à réaliser, la majorité des machines parallèles appartiennent à cette classe.

-CC-NUMA (Cache-Coherent NUMA) : Type d'architecture NUMA intégrant des caches.

-COMA (Cache Only Memory Access) : Les mémoires locales se comportent comme des caches de telle sorte qu'une donnée n'a pas de processeur propriétaire n'y d'emplacement déterminé en mémoire.

-OSMA (Operating System Memory Access) : Le système d'exploitation gère les accès aux données distantes, traite les défauts de pages au niveau logiciel puis également les requêtes d'envoi/copie de pages distantes

Schéma récapitulatif avec les exemples de machines existantes pour chaque classe de machines.

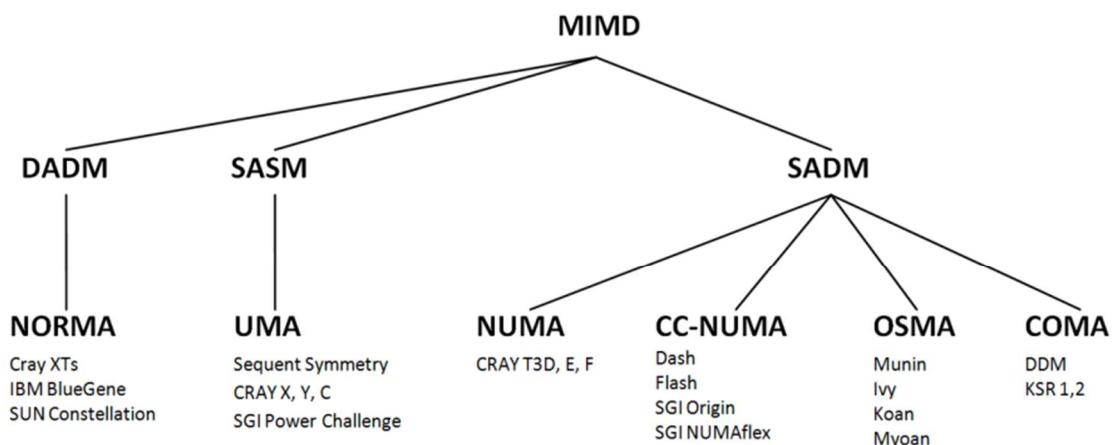


Figure 6: Classification MIMD de Raina

En dehors de ses deux classifications énumérées plus haut (taxonomie de Von, classification de RAINA), il existe aussi d'autres manières de classer les modèles parallèles tels que :

-La classification suivant la topologie du réseau d'interconnexion : Ce qui distingue les machines parallèles dans cette classification est la topologie du réseau d'interconnexion utilisés pour relier les différents nœuds de calculs avec les unités de mémoires. Elle est très importante dans la conception des algorithmes parallèles car elle peut diriger la tâche de conception ou influencer le temps d'exécution globale du programme. On distingue principalement deux classes de réseaux d'interconnexion :

-Les réseaux à topologie statique : Ce sont des réseaux dont la topologie est définie une fois pour toute lors de la construction de la machine parallèle. Elles sont utilisées dans les machines à passage de message et les machines cellulaires, etc.

-Les réseaux à topologie dynamiques : Ce sont des réseaux dont la topologie (topologie logique ou comportement du réseau) peut varier au cours de l'exécution d'un programme parallèle ou entre deux exécutions de programmes. Les réseaux dynamiques sont souvent utilisés dans les multiprocesseurs à mémoire partagés.

-La classification suivant la granularité du calcul : La taille du grain d'une machine parallèle peut se définir par la longueur typique d'un morceau de code exécutable sans interruption par l'unité de calcul [Was94]. Une erreur peut être externe provoquée par exemple par l'ordonnanceur sur système d'exploitation, ou bien interne, provoquée par le processus lui-même suite à un besoin de communication avec d'autres processus (ou processeurs). Selon la taille du grain de calcul, on rencontre trois classes de machine parallèle :

– Les machines à grains fins : Pour ces modèles, on suppose que le nombre de processeurs est sensiblement égal au nombre de données en entrée. Les processeurs exécutent de très petits morceaux de code entre deux communications successives. Le coût des communications est négligé au profit des calculs. Cette classe correspond aux machines systoliques ;

– Les machines à gros grains : Pour ces modèles, la taille de chaque mémoire locale est beaucoup plus grande que la taille d'une donnée. Et donc, chaque processeur est capable d'exécuter sans interruption des procédures voir des programmes entiers. Ces modèles reconnaissent et prennent en compte le coût des communications sans pour autant spécifier la topologie du support de communication.

– Les machines à grains moyens : Ce sont les machines qui sont entre les deux classes précédentes.

4. Calcul de la complexité parallèle

Pour étudier les performances d'un algorithme parallèle, plusieurs mesures sont utilisées. La plus naturelle est le temps d'exécution de l'algorithme. Il correspond au temps écoulé entre le moment où le premier processeur commence l'exécution de l'algorithme et le moment où le dernier processeur finit cette exécution. Dans la suite de cette section, le temps d'exécution de l'algorithme parallèle sur p processeurs sera noté T_p . Ce temps peut être aussi défini comme étant la somme du temps de calculs, du temps de communication et du temps d'attente d'un processeur arbitraire j

$(T_p = T_{\text{calculs}}^j + T_{\text{coms}}^j + T_{\text{attente}}^j)$, ou comme étant la somme de ces trois temps sur tous les processeurs divisés par le nombre de processeurs

$(T_p = 1/p(\sum_{i=1}^p T_{\text{calculs}}^i + \sum_{i=1}^p T_{\text{coms}}^i + \sum_{i=1}^p T_{\text{attente}}^i))$.

Une manière simple d'évaluer les communications est de donner le nombre de messages transmis et leur taille en octets. Ainsi, envoyer un message ayant k octets prendra un temps

$T_{\text{msg}} = T_{\text{init}}^j + k \times T_{\text{trans}}^j$, où T_{init}^j est le temps d'initialisation de la communication, et T_{trans}^j est le temps pour transférer un octet entre deux processeurs. La réalité est souvent plus complexe car il faut prendre en compte la nature du réseau. La latence indique le temps nécessaire pour communiquer un message unitaire entre deux processeurs. Tout envoi d'un message prendra donc

un temps supérieur à celui de la latence. Le facteur d'accélération (« speedup ») correspond au rapport entre le temps d'exécution de l'algorithme séquentiel optimal T_s et le temps d'exécution de l'algorithme parallèle sur p processeurs : $A_p = T_s/T_p$. Aussi, p est le meilleur facteur d'accélération que l'on puisse obtenir.

L'efficacité est le facteur d'accélération divisé par le nombre de processeurs : $E_p = A_p/p$

Il représente la fraction de temps où les processeurs travaillent. Il caractérise aussi l'efficacité avec laquelle un algorithme utilise les ressources de calcul d'une machine.

On s'intéresse aussi à l'extensibilité (« scalability ») lorsqu'on regarde l'évolution du temps d'exécution et du facteur d'accélération en fonction du nombre de processeurs.

5. Modèles de calcul parallèle

On appelle modèle de calcul parallèle « la description d'une machine parallèle destinée à être utilisée par les concepteurs d'algorithmes parallèles ». Le choix d'un modèle est très souvent guidé par son utilisation sous-jacente dans le but d'obtenir des résultats concluants sur une ou un ensemble de machines données.

Un bon modèle de calcul parallèle doit avoir un certain nombre de propriétés dont les plus importantes sont les suivantes :

- La simplicité de compréhension : c'est-à-dire ne pas être ambigu et avoir un minimum de paramètres ;
- La simplicité d'utilisation : c'est-à-dire qu'il ne doit pas augmenter la complexité du problème à résoudre en rappelant par exemple les petits détails inhérents à l'architecture des machines ;
- La bonne définition : c'est-à-dire qu'il doit être bien et complètement défini pour pouvoir servir de plate-forme commune pour différents développeurs ;
- La généralité : il doit refléter les caractéristiques de la majorité des architectures existantes ;
- La prédiction (représentation) des performances : les prédictions théoriques des performances des algorithmes doivent être en adéquation avec leur exécution sur les machines réelles. Autrement dit, les bons algorithmes pratiques ne doivent résulter que de bons algorithmes théoriques ;
- La faisabilité (réalité) : La machine abstraite décrite dans un bon modèle doit être réalisable par les technologies en cours sans la violation d'un grand nombre des hypothèses du modèle ;
- La robustesse dans le temps : un bon modèle doit survivre à l'évolution des technologies utilisées par les machines parallèles réelles qu'il décrit ;
- La portabilité : les algorithmes conçus suivant le modèle ne doivent pas être dépendant des architectures des machines.

Il est malheureusement difficile de concevoir un modèle répondant aux critères sus exprimés. Le modèle séquentiel de Von Neumann satisfait en grande partie ces différentes caractéristiques, c'est d'ailleurs pour cette raison qu'il est utilisé sur la plupart des machines actuelles. Il modélise correctement les machines, ce qui permet de réaliser de bonnes prédictions, mais sous une restriction : il suppose la mémoire des machines infinie, ce qui n'est pas le cas en pratique. Lorsque la mémoire est saturée, les performances se dégradent très fortement et les prédictions effectuées ne sont plus valables. Von Neumann avait de même mis en avant la hiérarchie mémoire d'un grand nombre de plates-formes actuelles qui va des caches aux bandes en passant par la mémoire et les disques, tout en soulevant les difficultés pour proposer un modèle prenant en compte cette hiérarchie mémoire. C'est néanmoins le modèle séquentiel le plus employé et probablement le plus performant. Dans le but de palier aux différentes insuffisances liées au modèle séquentiel, des

modèles parallèles ont vu le jour mais aucun d'eux ne s'est imposé jusqu'ici de la même façon que le modèle séquentiel de Von Neumann. Beaucoup de modèles proposés sont loin de répondre aux différentes caractéristiques énoncées ci-dessus.

6. Quelques modèles de calcul parallèle

Nous présenterons dans cette section différentes grandes classes de modèles de calcul parallèle. Nous présenterons quelques modèles à grain fin parmi lesquels le modèle PRAM (Parallel Random Access Memory), le modèle systolique, le modèle de grille à deux dimensions et l'hypercube.

a) Le modèle PRAM (Parallel Random Access Memory)

Il comprend un ensemble de processeurs possédant chacun une mémoire locale de petite taille (égale à $O(1)$), et une mémoire partagée par laquelle les processeurs peuvent communiquer. Les processeurs travaillent de manière synchrone, et chacun peut accéder à n'importe quelle place de la mémoire partagée en temps constant afin de lire ou d'écrire une donnée.

Le modèle PRAM est simple et est très utile pour le parallélisme des problèmes étudiés. Il constitue souvent une première étape pour la parallélisation. En vertu de son haut niveau d'abstraction, il permet souvent de savoir si un problème peut être parallélisé ou non et dans quelle mesure. La description des algorithmes est très simple, car il suffit de décrire une séquence d'opérations parallèles exécutées par les processus sans se préoccuper des communications entre les processeurs.

L'inconvénient de ce modèle est qu'il est fortement éloigné des machines réelles. La plupart des machines sont à mémoire distribuée et non à mémoire partagée et les contraintes technologiques pour que beaucoup de processeurs puissent accéder en temps constant à une mémoire commune sont telles qu'aucune machine PRAM n'a encore vu le jour. Par conséquent il faut souvent réadapter l'algorithme PRAM à la structure de la machine choisie. Ce modèle reste néanmoins d'actualité et un certain nombre d'extensions ont été proposées.

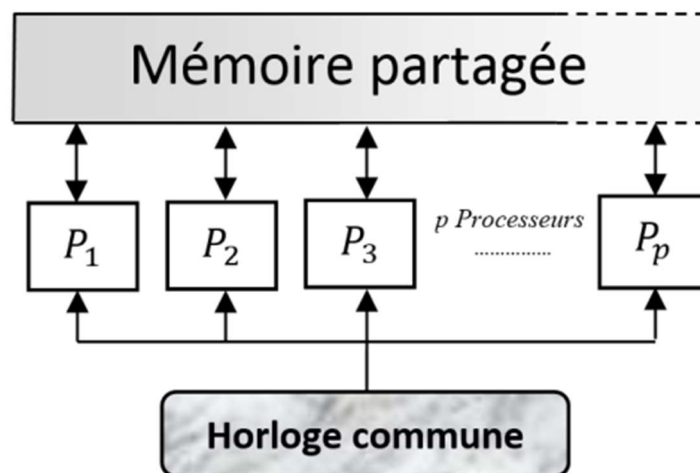


Figure 7: Modèle PRAM

b) Le Modèle systolique

Le concept d'automate cellulaire est étroitement lié au concept de « réseaux systolique ». On peut définir un réseau systolique comme un réseau de processeurs qui calculent et échangent des données régulièrement. L'analogie est souvent faite avec la régularité de la contraction cardiaque

qui propage le sang dans le système circulatoire du corps. Chaque processeur d'un réseau systolique peut être vu comme un cœur jouant le rôle de pompe sur plusieurs flots le traversant. Le rythme régulier de ces processeurs maintient un flot de données constant à travers tout le réseau.

Une donnée introduite une seule fois dans le réseau est propagée d'un processeur à un processeur voisin et peut ainsi être utilisée un grand nombre de fois. Cette propriété autorise de gros débits de calculs même si la cadence des entrées-sorties reste faible. En d'autres termes, on évite les engorgements des buffers d'entrées-sorties. Ceci rend le modèle systolique adéquat pour beaucoup de problèmes. Ceux dont le nombre de calculs sur une même donnée est largement supérieur à son nombre d'entrées-sorties.

Les caractéristiques dominantes d'un réseau systolique peuvent être définies par un parallélisme massif et décentralisé, par des communications locales et régulières, et par un mode opératoire synchrone. Pour décrire un réseau systolique, il est donc nécessaire de spécifier :

- La topologie du réseau d'interconnexion des processeurs ;
- L'architecture d'un processeur (description des registres et canaux : nom, type, sémantique, etc.) ;
- Le programme d'un processeur ;
- Le flot de données consommées par le réseau pour produire une solution.

c) Modèle grille à deux dimensions et modèle hypercube

Parmi les modèles de machines à mémoires distribués, la grille à deux dimensions et l'hypercube ont été beaucoup utilisés [Fer96, Lei92]. Dans ces modèles chaque processeur a sa propre mémoire locale de taille constante, il n'existe pas de mémoire partagée. Les processeurs peuvent communiquer uniquement grâce à un réseau d'interconnexion. Comme dans le cas de PRAM et du modèle systolique, les processeurs travaillent de manière synchrone. À chaque étape, chaque processeur peut envoyer un mot de données à un de ses voisins, recevoir un mot de données d'un de ses voisins, et effectuer un traitement local sur ces données. La complexité d'un algorithme est définie comme étant le nombre d'étapes de son exécution.

Ces modèles prennent explicitement en compte la topologie du réseau d'interconnexion. Ce dernier présente différentes caractéristiques importantes comme :

- Le degré, qui est le nombre maximal de voisins d'un processeur, il correspond à une sorte de limitation architecturale donnée par le nombre maximum de liens physiques associés à chaque processeur ;
- Le diamètre, qui est la distance maximale entre deux processeurs (cette distance est donnée par le plus court chemin dans le réseau d'interconnexion entre les deux processeurs les plus éloignés). Il donne une borne inférieure sur la complexité des algorithmes dans lesquels deux processeurs arbitraires doivent communiquer ;
- La largeur, qui est le nombre minimum de liens à enlever afin de diviser le réseau en deux réseaux de même taille (plus ou moins un). Elle présente une borne inférieure sur le temps d'exécution des algorithmes où il existe une phase qui fait communiquer une moitié des processeurs avec une autre moitié.

La grille à deux dimensions

La grille à deux dimensions de taille p est composée de p processeurs $P_{i,j}$, $1 \leq i, j \leq \sqrt{p}$, tels que le processeur $P_{i,j}$ est relié aux processeurs $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$ et $P_{i,j+1}$ pour $2 \leq i, j \leq \sqrt{p} - 1$. La grille a un degré de 4, un diamètre de $O(\sqrt{p})$ et une largeur de bande de \sqrt{p} . Cette structure ainsi que ses dérivés, comme la grille à trois dimensions, le tore à 2 ou 3 dimensions (les processeurs à la périphérie sont connectés avec ceux se trouvant à l'opposé) ont l'avantage d'être simple et flexible.

Schéma d'une grille a deux dimensions de taille 16.

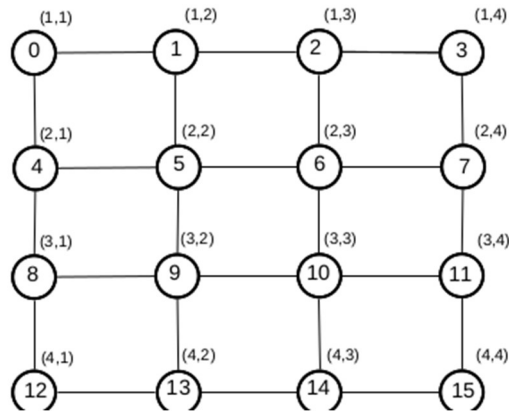
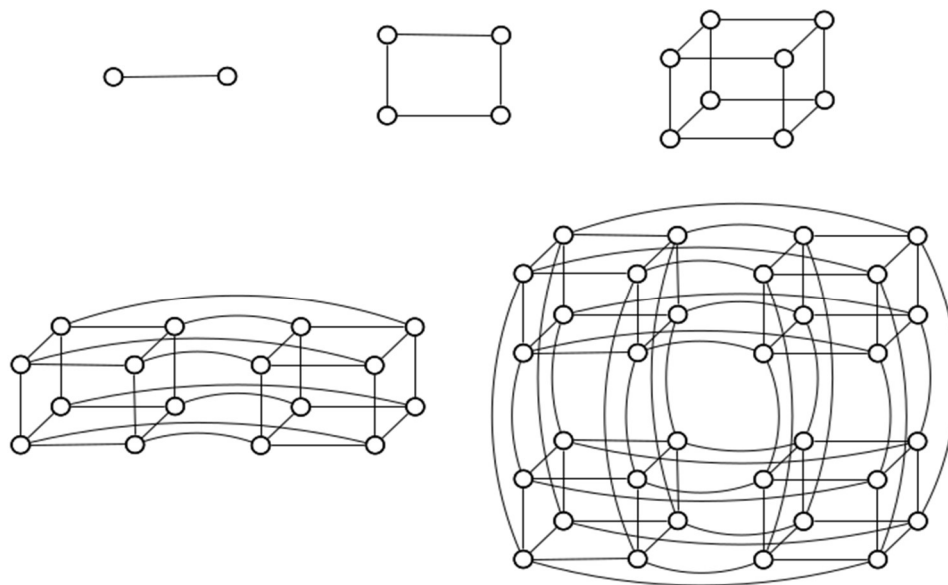


Figure 8: Grille a deux dimensions de taille 16

d) L'hypercube

L'hypercube de dimension d est composé de $p = 2^d$ processeurs, numérotés de 0 à $p-1$. Deux processeurs P_i et P_j sont reliés si et seulement si la représentation binaire de i et celle de j diffèrent seulement d'un bit. Si ce bit est égal à k , on dit que P_i et P_j sont voisins suivant la dimension k et que $j = i^k$. Le degré et le diamètre sont égaux à $\log p$, et la largeur de bande est égale à $p/2$. L'hypercube est attrayant de par sa régularité et son petit diamètre, mais est difficilement réalisable en pratique (à cause du degré qui augmente avec le nombre de processeurs). Le schéma ci dessus présente des hypercubes de dimensions 2, 4, 8, 16 et 32.

Pour écrire un algorithme dans un de ces modèles à mémoire distribuée, il est possible, soit de simuler un algorithme PRAM, soit d'écrire un algorithme spécifique pour l'architecture du modèle choisi. La première solution bien qu'élégante conduit souvent à des algorithmes non efficaces. En revanche, la deuxième solution permet d'obtenir des résultats efficaces mais reste très souvent compliquée. De plus, la plupart du temps, elle ne permet pas d'écrire des algorithmes portables, puisque chaque modèle à mémoire distribuée dépend fortement de la topologie du réseau choisi. Par conséquent, lorsqu'on change la topologie du réseau d'interconnexion, il faut souvent changer d'algorithme.



La mise en œuvre d’algorithmes parallèles avec les modèles parallèles existants a montré un certain nombre de compromis :

- Efficacité versus Portabilité des algorithmes : modèle hypercube - modèle textsc-pram ;
- Généralité versus Réalité : modèle systolique - modèle PRAM ;
- Simplicité de programmation versus Efficacité d’exécution : modèle à mémoire partagée -modèle à mémoire distribuée ;
- Simplicité d’analyse versus Simplicité de réalisation : modèle synchrone – modèle asynchrone.

Il existe plusieurs api (application Programming interface) qui permettent de faire des programmations en parallèle. Entre autres on peut citer :

- OPENMP (Open Multi-Processing) : qui est une interface de programmation pour calcul parallèle sur architecture à mémoire partagée ; c’est donc une api de programmation multi threads.
- OPENCL (Open Computing Language) : c’est la combinaison d’une api et d’un langage de programmation dérivé du C. Il est conçu pour programmer des systèmes parallèles hétérogène comprenant à la fois un CPU (Central Processing Unit) multicœurs et d’un GPU (Graphics Processing Unit).
- MPI (Message Passing Interface) : c’est une norme conçue pour le passage de message entre ordinateurs distants ou dans un ordinateur multiprocesseur. Elle est devenue un standard de communication pour des nœuds exécutant des programmes parallèles sur des système à mémoire distribuée. Mais dans la suite de notre travail nous allons nous attarder sur OPENMP.

II. OPENMP

1. Définition et Principe de fonctionnement

a) Définition

OPENMP (Open Multi-Processing) est une interface de programmation à mémoire partagée basé sur les directives à insérer dans le code source (C, C++, Fortran). On peut aussi dire que c’est un ensemble de directives de compilation et de bibliothèques de fonctions facilitant la création des programmes multi-threads.

Ainsi, le calcul parallèle est donc vu comme la répartition des charges de calcul sur plusieurs processus légers appelé threads. Le but du calcul parallèle étant de diminuer le temps d'exécution du programme.

b) Principe Fonctionnement

❖ Concepts Généraux

- Ce processus active des processus légers(threads) à l'entrée d'une région parallèle.
- Chaque processus léger exécute une tâche composée d'un ensemble d'instructions.
- Pendant l'exécution d'une tâche une variable peut être lue et/ou modifiée en mémoire. Elle peut être définie dans la pile (stack) (espace mémoire local) d'un processus léger ; on parle alors de variable privée.
Elle peut être définie dans un espace mémoire partagé.
- Un programme OpenMP est une alternance de région séquentielle et de région parallèle.
- Une région séquentielle est toujours exécutée par la tâche maître, celle donc le rang est 0.
- Une région parallèle peut être exécutée par plusieurs tâches à la fois.
- Les tâches peuvent se partager le travail se trouvant dans la région parallèle.
- La répartition du travail consiste à :
 - Exécuter une boucle par répartition des itérations entre les tâches ;
 - Exécuter plusieurs sections de code mais une seule par tâche ;
 - Exécuter plusieurs occurrences d'une même procédure par différentes tâches (orphaning).
- Généralement, les tâches sont affectées aux processeurs par le système d'exploitation.

❖ Construction d'une région parallèle (fork/join)

Le principe de fonctionnement OpenMP est ce que l'on appelle le modèle « Fork-join ». En effet, à l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle <<Fork and Join>>. A l'entrée de la région la tâche maître crée/active(fork) des processus fils (processus légers) qui disparaissent(s'assoupissent) en fin de région parallèle(join) pendant que la tâche maître poursuit seule l'exécution jusqu'à l'entrée de la région parallèle suivante.

La création d'une région parallèle se fait grâce à la ligne suivante :

pragma omp parallel (en C et C++)

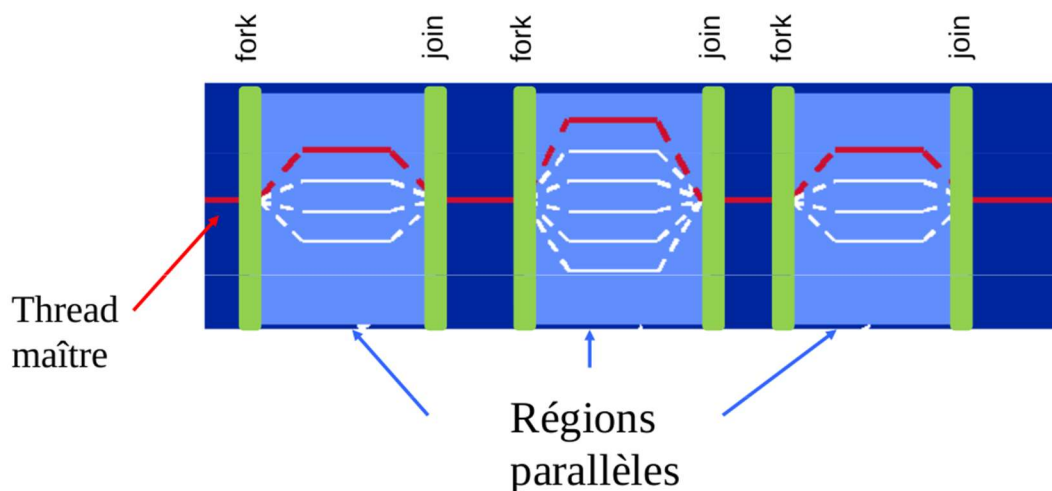
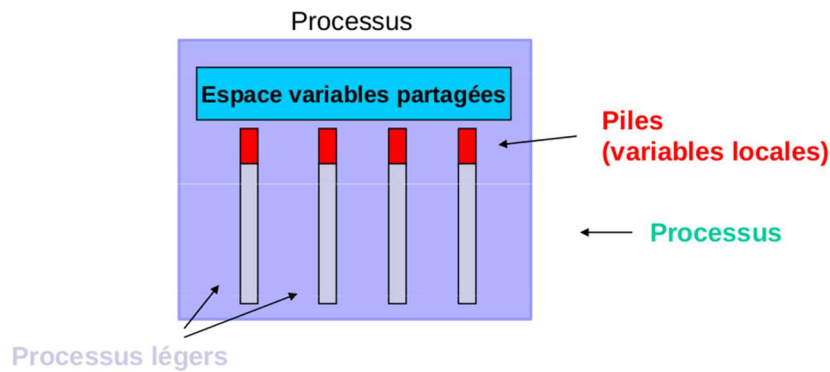


Figure 9: Schéma illustratif régions parallèle

Les threads accèdent aux mêmes ressources que le processus. Elles ont une pile (stack, pointeur de pile et pointeur d'instructions propres).



2. Structure d'un programme OpenMP

Un programme OpenMP est constitué de trois parties principales :

- Directives et clauses de compilation ;
 - Elles servent à définir le partage du travail, la synchronisation et le statut privé ou partage de 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.



La forme générale d'une directive OpenMP est la suivante :

Sentinelles directives [clause[clause]...]

Cette ligne est ignorée par le compilateur si l'option interprétation de la directive OpenMP n'est pas spécifiée. La sentinelle est une chaîne de caractère donc la valeur dépend du langage utilisé. Les fonctions OpenMP se trouvent toujours dans une région parallèle. On peut définir plusieurs régions parallèles dans un programme à l'aide de la directive suivante :

```
pragma omp parallel  
{  
...  
}
```

3. Installation d'OpenMP

a) Installation d'OpenMP sur windows

Cette partie concerne les utilisateurs utilisant Code::block comme IDE.

Quel que soit la version choisie, le MinGW intégré ne supporte pas OpenMP. Vous allez dans le répertoire `C:\Program Files (x86)\CodeBlocks` et changer le répertoire MinGW en MinGW_old, pour revenir en arrière en cas de besoin.

- Vous allez chercher le MinGW de TDM disponible ici : <http://tdm-gcc.tdragon.net/download>.
- Vous lancez l'application et choisissez CREATE. Et vous choisissez 32 ou 64 bits.
- On veut que le TDM-gcc vienne remplacer l'ancien wingw, on remplit la destination comme suit : `C:\Program Files (x86)\CodeBlocks\MinGW\`
- Il faut ne pas oublier de cocher que l'on veut installer OpenMP.

On finit l'installation de TDM-gcc.

- On lance code::blocks. Il faut ensuite aller changer les noms des exécutables pour la compilation : Settings -> Compiler.

Et aller dans l'onglet (image ci-dessous) : Global compiler settings, et sous onglet toolchain executables. Entrez les exécutables correspondant au TDM installé (varie selon 32 ou 64 bits). Tout est ok pour la partie installation de TDM-gcc.

b) Installation d'OpenMP sur linux

❖ Avec Code::Blocks

Pour pouvoir utiliser OpenMP avec code block sous Ubuntu, il faut :

- Installer code block à l'aide de la commande : **sudo apt-get install codeblocks**
- Installer ensuite OpenMP GNU GCC avec la commande **sudo apt-get install libgomp1** (pour les 32 bits) ou **sudo apt-get install lib64gomp1** (pour les 64 bits). Vous pouvez maintenant créer un nouveau projet et tester. Mais avant la compilation, il va falloir indiquer que nous utilisons OpenMP et où se trouve la librairie.
- Dans le menu, **Project -> Build options**, aller dans l'onglet **Compiler Settings**, sous onglet **Other options** et mettre **-fopenmp**. Enfin, aller dans l'onglet **Linker Settings** et cliquer sur **add** pour indiquer où se trouve **libgomp1.so**. Pour trouver **libgomp.so**, dans un terminal, tapez : **locate libgomp**. Prendre la librairie se situant dans votre dernière version de GCC.

❖ En ligne de commande

- Au préalable se rassurer que vous avez le compilateur gcc installé en tapant la commande suivante : **gcc -v** ou **gcc -version**

Dans le cas contraire installer le compilateur gcc au travers de la commande :

sudo apt install gcc

- En suite configurer OpenMP sur votre machine s'il est installé au travers de la commande : **echo |cpp -fopenmp -dM |grep -i open**

Si OpenMP n'est pas installé par défaut sur votre machine vous pouvez l'installer au travers la commande :

```
sudo apt install libomp-dev
```

(1) Compilation (Compilation avec GNU)

Pour compiler un programme parallélisé en OpenMP il faut définir le nombre de threads avec la variable d'environnement OMP_NUM_THREADS. Par exemple pour utiliser 4 threads :

```
$export OMP_NUM_THREADS=4.
```

La compilation d'un programme OPENMP se fait grâce à la commande suivante :

```
gcc -o nomprogramme -fopenmp nomprogramme.c
```

(2) Exécution

Après la compilation l'exécution du programme se fait grâce à la commande :

```
./nomprogramme
```

4. Variables avec OpenMP

a) Portée des variables

- **Les variables statiques et dynamique**

Statique : L'emplacement en mémoire est défini dès sa déclaration par le compilateur.

Automatique : L'emplacement en mémoire attribue au lancement de l'unité du programme ou elle est déclarée (existence garantie que pendant l'exécution de l'unité).

- **Variables globales**

La variable globale déclarée au début du programme principal, elle est statique.

- *initialisées à la déclaration (exemple : parameter, data)

- *non-initialisées à la déclaration (exemple : en Fortran Common, en C les variables d'unités de fichier, les variables externes static).

- **Variables locales**

Variables à portée restreinte à l'unité ou elle est déclarée, 2 catégories :

- * Variables locales automatiques

- * Variables locales rémanentes (statiques) si elles sont :

- Initialisés implicitement à la déclaration ;

- Déclarées par une instruction de type DATA

- Déclarées avec l'attribut SAVE → valeur conservée entre deux appels.

b) Statuts d'une variable

Le statut d'une variable dans une zone parallèle est :

- **SHARED** : Elle se trouve dans la mémoire globale ;
- **PRIVATE** : Elle est la pile de chaque thread, sa valeur est définie à l'entrée de la zone.

Schéma illustratif du statut **PRIVATE** :

```

1 //OpenMP header
2 #include <omp.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(int argc, char* argv[])
7 {
8     int nthreads=10, tid;
9
10    //begin of parallel region
11    #pragma omp parallel private(nthreads)
12    {
13        //getting thread number
14        tid = omp_get_thread_num();
15        printf("Welcome to GFG from thread = %d\n" , tid);
16        if(tid == 0){
17            //only master thread does his
18            nthreads = omp_get_num_threads();
19            printf("Number of threads = %d\n" , nthreads);
20        }
21    }
22    printf("Number of threads = %d\n" , nthreads);
23 }
24

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ export OMP_NUM_THREADS=8
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o firstopenmp -fopenmp first
openmp.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./firstopenmp
Welcome to GFG from thread = 2
Number of threads = 8 d'un programme OPENMP se fait grace a la commande suiv
Welcome to GFG from thread = 3 emp nonprogramme.c
Welcome to GFG from thread = 5
Welcome to GFG from thread = 7
Welcome to GFG from thread = 0 execution du programme se fait grace a la comm
Welcome to GFG from thread = 4
Welcome to GFG from thread = 1
Welcome to GFG from thread = 6
Number of threads = 10

```

Figure 11: Résultat du programme de la figure 10

Figure 10: Utilisation de la clause private (nthreads)

Déclaration du statut d'une variable :

-C et C++

#pragma omp parallel private(list_var)

c) Clause de la directive parallèle

La directive PARALLEL dispose de plusieurs clauses, et on peut citer entre autres :

- NONE : equivalent de l'IMPLICIT NONE en fortran. Toute variables devra avoir un statut definie explicitement.
- SHARED(list_var) : Variables partagés entre les threads.
- PRIVATE(list_var) : Variable privée a chaqu'une des threads, indefinies en dehors du bloc PARALLEL
- FIRSTPRIVATE(list_var) : Variable initialisée avec la valeur que la variable d'aorigine avait juste avant la section parallele.

Illustration :

Dans le programme suivant, la variable 'x' est déclarée comme variable globale et initialisée à 4. A l'entrée de la région parallèle elle est déclarée firstprivate et sa valeur d'initialisation est conservée. Dans la région parallèle on fait une incrémentation qui ajoute chaque fois le numéro du thread s'exécutant à 'x' ; donc le thread de numéro 0 affichera 4, car l'opération effectuée sera $4+0=4$.

```

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

int main()
{
    int x=4, nbthreads;
    #pragma omp parallel firstprivate(x)
    {
        //printf("affiche de valeur: %d\n", x);
        nbthreads = omp_get_thread_num();
        x = x + nbthreads;
        printf("affiche de la valeur:%d\n", x);
    }
}

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o fisrpriv -fopenmp fisrpriv.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./fisrpriv
affiche de la valeur:6
affiche de la valeur:4
affiche de la valeur:7
affiche de la valeur:10
affiche de la valeur:11
affiche de la valeur:5
affiche de la valeur:9
affiche de la valeur:8
hermann@hermann-HP-ProBook-640-G1:~/Bureau$

```

- **DEFAULT**

Le statut par défauts des variables peut être changé avec la clause **default()** ; Cette clause prend comme argument: private, firstprivate, shared ou none en fortran et shared ou none en C/C++. default(none) permet à l'utilisateur de spécifier le statut de toute les variables utilisées dans la région parallèle.

Illustration :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main()
6 {
7     int x=4, y=3, nbthreads;
8     #pragma omp parallel default(none) private(nbthreads, x, y)
9     {
10        //printf("affiche de valeur: %d\n", x);
11        nbthreads = omp_get_thread_num();
12        x = x + nbthreads;
13        y = y + nbthreads;
14        printf("affiche de la valeur x= %d et y= %d\n", x, y);
15    }
16    printf("Dans la region sequentielle x= %d et y= %d\n", x, y);
17 }
18
19

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o fisrpriv -fopenmp fisrpriv.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./fisrpriv
affiche de la valeur x= 3 et y= 3
affiche de la valeur x= 2 et y= 2
affiche de la valeur x= 1 et y= 1
affiche de la valeur x= 0 et y= 0
Dans la region sequentielle x= 4 et y= 3
hermann@hermann-HP-ProBook-640-G1:~/Bureau$

```

- **LASTPRIVATE**

Elle permet à une variable de garder la dernière valeur qu'elle a eu dans la région parallèle. Elle n'est utilisable qu'avec les directives **do**, **for** et **section**.

ILLUSTRATION :

Ce bout de code permet de crée une variable globale 'x' et de modifier sa valeur dans la région parallèle après l'avoir déclarée LASTPRIVATE dans cette région et elle ressort de là avec la dernière valeur qu'elle a reçu dans cette région.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main()
6 {
7     int x, nbthreads;
8     #pragma omp parallel
9     {
10    #pragma omp for lastprivate(x)
11    for(int i=0; i<10; i++)
12    x=i;
13    printf("La valeur de X= %d\n", x);
14
15 }
16 printf("La valeur dans le code sequentielle X= %d\n", x);
17
18 }

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o lastpriv -fopenmp lastpriv.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./lastpriv
La valeur de X= 9
La valeur de X= 9
La valeur de X= 9
La valeur de X= 9
La valeur dans le code sequentielle X= 9
hermann@hermann-HP-ProBook-640-G1:~/Bureau$

```

d) Étendue d'une région parallèle

L'étendue d'une construction OPENMP représente le champ d'influence de celle-ci dans le programme. L'influence (la portée) d'une région parallèle s'étend aussi bien au code contenu lexicalement dans cette région (étendue statique), qu'au code des sous programmes appelés. L'union des deux représente « L'étendue dynamique ».

Illustration par un bout de code :

```

1 #include <stdio.h>
2 #include <omp.h>
3
4 void sub(void)
5 {
6     int p=0;
7     #ifdef _OPENMP
8     p = omp_in_parallel();
9     #endif
10 }
11 printf("Parallele ? : %d\n", p);
12 }
13 int main()
14 {
15     void sub(void);
16     #pragma omp parallel
17     {
18         sub();
19     }
20     return 0;
21 }

```

-Transmission par argument : Dans une procédure, tous les valeurs transmises par argument(dummy parameter) par référence, héritent du statut défini dans l'étendue (lexicale) de la région.

-L'allocation dynamique : L'opération d'allocation/désallocation de mémoire peut être fait à l'intérieur d'une région parallèle. Si elle porte sur une variable privée elle sera locale a chaque tâche; si elle porte sur une variable partagée il est alors plus prudent que seul une tâche se charge de cette opération.

e) Compléments

La construction d'une région parallèle admet deux autres clauses à savoir :

-RÉDUCTION : pour les opérations de réduction avec synchronisation implicite entre les tâches ;

-NUM_THREADS : elle permet de spécifier le nombre de tâches souhaité à l'entrée d'une région parallèle de la même manière que le ferait le sous-programme OMP_SET_NUM_THREADS. D'une région parallèle à l'autre, le nombre de tâches concurrentes peut être variable si on le souhaite. Pour cela, il suffit d'utiliser le sous-programme OMP_SET_DYNAMIC ou de positionner la variable d'environnement OMP_DYNAMIC à « true ».

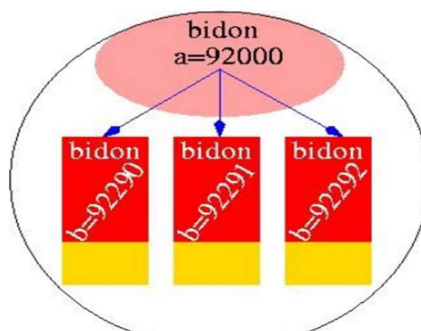
Il est possible d'imbriquer (nesting) des régions parallèles, mais cela n'a d'effet que si ce mode a été activé à l'appel du sous-programme OMP_SET_NESTED ou en positionnant la variable d'environnement OMP_NESTED.

Illustration : Le code suivant permet de définir une région parallèle avec une variable privée qui s'exécutera sur 3 threads. A chaque fois le numéro du thread (rang) qui s'exécute est affiché, et à l'intérieur de cette même région on définit une nouvelle région parallèle avec toujours la même variable privée qui s'exécutera cette fois si sur un seul thread et affiche aussi le numéro du thread.

```
1 #include <stdio.h>
2 #include <omp.h>
3 int main()
4 {
5     int rang;
6     #pragma omp parallel private(rang) num_threads(3)
7     {
8         rang=omp_get_thread_num();
9         printf("Mon rang dans region 1 : %d \n",rang);
10    #pragma omp parallel private(rang) num_threads(1)
11    {
12        rang=omp_get_thread_num();
13        printf(" Mon rang dans region 2 : %d \n",rang);
14    }
15 }
16 return 0;
17 }
```

```
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o etendue -fopenmp etendue.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ export omp_dynamic=true
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ export mp_nested=true
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./etendue
Mon rang dans region 1 : 0
Mon rang dans region 2 : 0
Mon rang dans region 1 : 2
Mon rang dans region 2 : 0
Mon rang dans region 1 : 1
Mon rang dans region 2 : 0
hermann@hermann-HP-ProBook-640-G1:~/Bureau$
```

NB : Comme nous l'avons dit plus haut les variables statiques sont par défaut partagées. Mais l'utilisation de la directive threadprivate permet de privatiser une instance statique (pour les threads et non les tâches...) et faire que celle-ci soit persistante d'une région parallèle à une autre. Si, en outre, la clause COPYIN est spécifiée alors la valeur des instances statiques est transmise à tous les threads.



5. Partage du travail

En principe, la construction d'une région parallèle et l'utilisation de quelques fonctions OpenMP suffisent à eux seuls pour paralléliser une portion de code. Mais il est, dans ce cas, à la charge du programmeur de répartir aussi bien le travail que les données et d'assurer la synchronisation des tâches. Heureusement, OpenMP propose trois directives (DO (équivalent au for en C), SECTIONS et WORKSHARE) qui permettent aisément de contrôler assez finement la répartition du travail et des données en même temps que la synchronisation au sein d'une région parallèle.

Par ailleurs, il existe d'autres constructions OpenMP qui permettent l'exclusion de toutes les tâches à l'exception d'une seule pour exécuter une portion de code située dans une région parallèle.

a) Les directives du partage du travail

Elles sont incluses dans les régions parallèles, doivent être rencontrées par tous les threads ou aucun. Elles repartissent le travail entre différents threads et impliquent une barrière à la fin de la construction (sauf si la clause NOWAIT a été spécifier).

A- La directive for

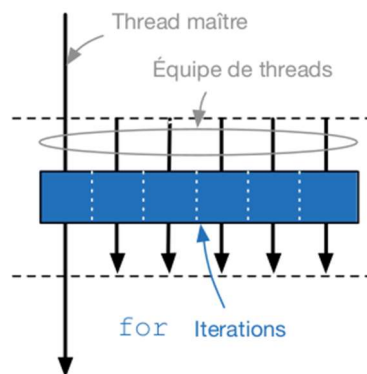
Elle repartit les itérations d'une boucle parallèle.

- Indique que les itérations de la boucle qui suit la directive doivent être exécutées en parallèle par l'équipe de threads.
- La variable d'itération est privée par défaut.
- Les boucles doivent avoir une forme itérative simple ; Les bornes doivent être les mêmes pour tous les threads et les boucles infinies ou while ne sont pas supporter.
- Clauses possible: schedule, ordered, private, firstprivate, lastprivate, reduction, collapse, nowait.

Format général de la directive :

```
#pragma omp for [ clause [ clause]... ]  
for(...)  
{  
...  
}
```

Illustration : Produit scalaire avec la clause **reduction**




```

1 #include <stdio.h>
2 #include <omp.h>
3 #define SIZE 5
4 int main () {
5     double sum , a[ SIZE ], b[ SIZE ];
6     // Initialization
7     sum = 0.;
8     for (size_t i = 0; i < SIZE ; i ++ ) {
9         a[i] = i * 0.5;
10        b[i] = i * 2.0;
11    }
12    // Computation
13    #pragma omp parallel for reduction(+: sum )
14    for (size_t i = 0; i < SIZE ; i ++ ) {
15        sum = sum + a[i ]* b[i ];
16    }
17    printf("la somme est sum= %g\n", sum);
18 }
19 printf ( " sum = %g\n" , sum );
20 return 0;
21 }

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o tor -fopenmp tor.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./tor
la somme est sum= 0
la somme est sum= 1
la somme est sum= 9
la somme est sum= 4
la somme est sum= 16
sum = 30
hermann@hermann-HP-ProBook-640-G1:~/Bureau$

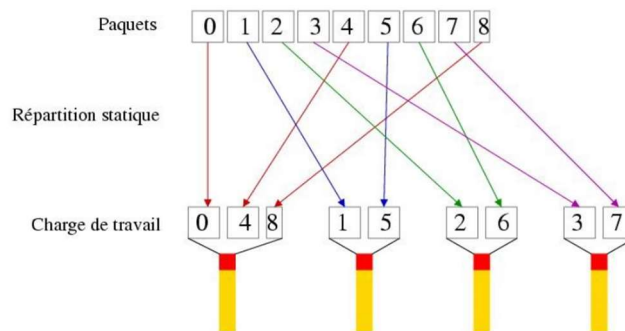
```

(1) Clause schedule

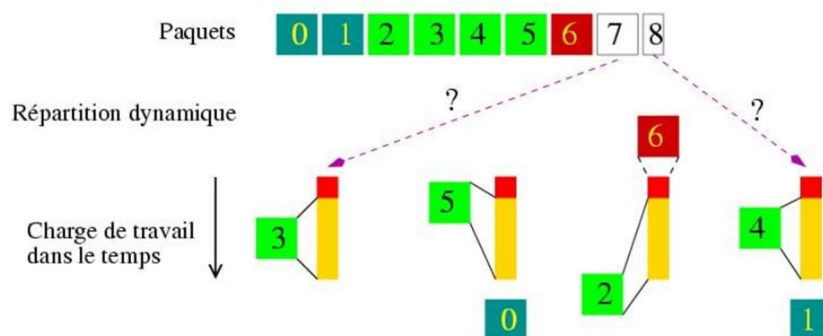
Elle spécifie la politique du partage des itérations : **schedule (type [, chunk])**

Cinq types sont possibles :

-static : consiste à diviser les itérations en paquets d'une taille donnée (sauf peut-être pour le dernier). Il est ensuite attribué, d'une façon cyclique à chacune des tâches, un ensemble de paquets suivant l'ordre des tâches jusqu'à concurrence du nombre total de paquets.



-dynamic : les itérations sont divisées en paquets de taille donnée. Sitôt qu'une tâche épuise ses itérations, un autre paquet lui est attribué.



-guided : les itérations sont divisées en paquets dont la taille décroît exponentiellement. Tous les paquets ont une taille supérieure ou égale à une valeur donnée à l'exception du dernier dont la taille peut être inférieure. Sitôt qu'une tâche finit ses itérations, un autre paquet d'itérations lui est attribué.

-runtime : Le choix de la politique est reporté au moment de l'exécution, par exemple par la variable d'environnement OMP_SCHEDULE.

-auto : Le choix de la politique est laissé au compilateur et/ou au runtime.

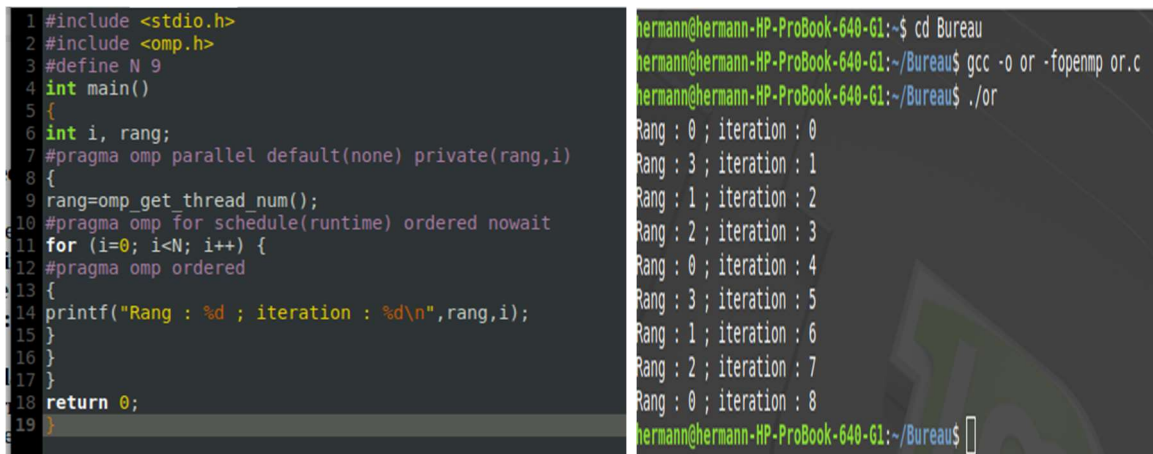
(2) La clause nowait

Elle permet aux threads qui finissent leur exécution en avance d'exécuter les autres instructions sans attendre les autres. Le programmeur doit juste s'assurer que la sémantique du programme est préservée.

(3) La clause ordered

Il est parfois utile (cas de débogage) d'exécuter une boucle d'une façon ordonnée. L'ordre des itérations sera alors identique à celui correspondant à une exécution séquentielle.

Illustration des deux clauses précédentes. Le programme suivant permet d'afficher le numéro du processus qui s'exécute (rang), avec le numéro d'itérations dans l'ordre et sans temps d'attente.



```
1 #include <stdio.h>
2 #include <omp.h>
3 #define N 9
4 int main()
5 {
6     int i, rang;
7     #pragma omp parallel default(none) private(rang,i)
8     {
9         rang=omp_get_thread_num();
10        #pragma omp for schedule(runtime) ordered nowait
11        for (i=0; i<N; i++) {
12            #pragma omp ordered
13            {
14                printf("Rang : %d ; iteration : %d\n",rang,i);
15            }
16        }
17    }
18    return 0;
19 }
```

```
hermann@hermann-HP-ProBook-640-G1:~$ cd Bureau
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o or -fopenmp or.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./or
Rang : 0 ; iteration : 0
Rang : 3 ; iteration : 1
Rang : 1 ; iteration : 2
Rang : 2 ; iteration : 3
Rang : 0 ; iteration : 4
Rang : 3 ; iteration : 5
Rang : 1 ; iteration : 6
Rang : 2 ; iteration : 7
Rang : 0 ; iteration : 8
hermann@hermann-HP-ProBook-640-G1:~/Bureau$
```

B- La directive sections

Une section est une portion de code exécutée par 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 tâches.

Toutes les directives SECTION doivent apparaître dans l'étendue lexicale de la construction SECTIONS.

Clauses possibles : private, firstprivate, lastprivate, reduction, nowait

Illustration :


```

1 int main()
2 {
3     int i, rang;
4
5     #pragma omp parallel private(rang) num_threads(2)
6     {
7         rang=omp_get_thread_num();
8         #pragma omp sections nowait
9         {
10             #pragma omp section
11             {
12                 printf("Tâche numéro %d : init. champ en X\n",rang);
13             }
14             #pragma omp section
15             {
16                 printf("tache numero %d: init. champ en Y\n", rang);
17             }
18         }
19     }
20     return 0;
21 }
22

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./sec
Tâche numéro 0 : init. champ en X
tache numero 1: init. champ en Y
hermann@hermann-HP-ProBook-640-G1:~/Bureau$

```

C-La directive SINGLE

La construction SINGLE permet de faire exécuter une portion de code par une et une seule tâche sans pouvoir indiquer laquelle. En général, c'est la tâche qui arrive la première sur la construction SINGLE mais cela n'est pas spécifié dans la norme. Toutes les tâches n'exécutant pas la région SINGLE attendent, en fin de construction END SINGLE, la terminaison de celle qui en a la charge, à moins d'avoir spécifié la clause NOWAIT.

Illustration : Dans notre exemple suivant, nous avons une région parallèle dans laquelle nous changeons la valeur de 'i' par une opération bien défini ; Et dans cette même région parallèle nous avons une région déclarée single dans laquelle nous modifions la valeur de 'i' en la faisant passer à 10.

Et après exécution nous voyons bien que la région SINGLE est exécutée par un seul thread (dans notre exemple c'est la tâche 1 qui a fait cette exécution)

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main()
6 {
7     int i, nbthread;
8     #pragma omp parallel private(i, nbthread)
9     {
10         nbthread = omp_get_thread_num();
11         i=2 + nbthread;
12         #pragma omp single
13         {
14             i=10;
15         }
16         printf("thread number %d avec i=%d\n", nbthread, i);
17     }
18     return 0;
19 }
20

```

```

hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o sin -fopenmp sin.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./sin
thread number 0 avec i=2
thread number 3 avec i=5
thread number 1 avec i=10
thread number 2 avec i=4
hermann@hermann-HP-ProBook-640-G1:~/Bureau$

```

Une clause supplémentaire admise uniquement par la directive de terminaison END SINGLE est la clause COPYPRIVATE. Elle permet à la tâche chargée d'exécuter la région SINGLE, de diffuser aux autres tâches la valeur d'une liste de variables privées avant de sortir de cette région.

D- La directive MASTER

La construction MASTER permet de faire exécuter une portion de code par la tâche maître seule. Cette construction n'admet aucune clause. Rappelons-le thread maître est le thread de numéro 0.

Illustration : Dans notre exemple suivant, nous avons une région parallèle dans laquelle nous changeons la valeur de 'i' par une opération bien défini ; Et dans cette même région parallèle nous avons une région déclarée MASTER dans laquelle nous modifions la valeur de 'i' en la faisant passée à 10.

Et après exécution nous voyons bien que la région MASTER est exécutée par le thread maître (il s'agit du thread d'indice 0).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main()
6 {
7     int i, nbthread;
8     #pragma omp parallel private(i, nbthread)
9     {
10        nbthread = omp_get_thread_num();
11        i=2 + nbthread;
12        #pragma omp master
13        {
14            i=10;
15        }
16        printf("thread number %d avec i=%d\n", nbthread, i);
17    }
18    return 0;
19 }
20
```

```
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o mas -fopenmp mas.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./mas
thread number 0 avec i=10
thread number 3 avec i=5
thread number 1 avec i=3
thread number 2 avec i=4
hermann@hermann-HP-ProBook-640-G1:~/Bureau$
```

b) Les directives de synchronisation

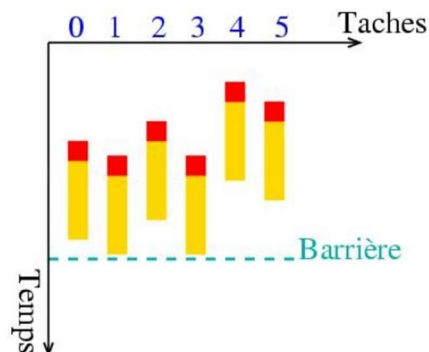
La synchronisation devient nécessaire dans les situations suivantes :

- Pour s'assurer que toutes les tâches concurrentes aient atteint un même niveau d'instruction dans le programme (barrière globale) ;
- Pour ordonner l'exécution de toutes les tâches concurrentes quand celles-ci doivent exécuter une même portion de code affectant une ou plusieurs variables partagées dont la cohérence (en lecture ou en écriture) en mémoire doit être garantie (exclusion mutuelle).
- Pour synchroniser au moins deux tâches concurrentes parmi l'ensemble (mécanisme de verrous).

(1) La directive BARRIER

La directive BARRIER synchronise l'ensemble des tâches concurrentes dans une région parallèle. Chacune des tâches attend que toutes les autres soient arrivées à ce point de synchronisation pour poursuivre, ensemble, l'exécution du programme.

Illustration : Dans notre exemple nous allons présenter le cas avec la directive barrier et le cas sans cette directive :



- Sans la directive :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main()
5 {
6     int i, j, nbthreads;
7     #pragma omp parallel private(nbthreads)
8     {
9         nbthreads=omp_get_thread_num();
10        #pragma omp single
11        {
12            i=5+ nbthreads;
13            j=5+ nbthreads;
14            printf("single:je suis le threads numero i=%d\n", nbthreads);
15        }
16        printf("paralle:je suis le threads numero j=%d\n", nbthreads);
17        #pragma omp master
18        {
19            printf("master:affiche %d\n", nbthreads);
20        }
21    }
22 }
23 }
```

```
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./bar1
single:je suis le threads numero i=0
paralle:je suis le threads numero j=3
paralle:je suis le threads numero j=0
master:affiche 0
paralle:je suis le threads numero j=1
paralle:je suis le threads numero j=2
hermann@hermann-HP-ProBook-640-G1:~/Bureau$
```

- Avec la directive : La partie situe dans la directive **barrier** va jouer le rôle de barrière a l'exécution des taches.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main()
5 {
6     int nbthreads;
7     #pragma omp parallel private(nbthreads)
8     {
9         nbthreads=omp_get_thread_num();
10        #pragma omp single
11        {
12            printf("single:je suis le threads numero %d\n", nbthreads);
13        }
14        printf("region parallele:je suis le threads numero %d\n", nbthreads);
15        #pragma omp barrier
16        #pragma omp master
17        {
18            printf("master:affiche %d\n", nbthreads);
19        }
20    }
21 }
22 }
```

```
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ gcc -o bar -fopenmp bar.c
hermann@hermann-HP-ProBook-640-G1:~/Bureau$ ./bar
single:je suis le threads numero 1
region parallele:je suis le threads numero 1
region parallele:je suis le threads numero 3
region parallele:je suis le threads numero 0
region parallele:je suis le threads numero 2
master:affiche 0
hermann@hermann-HP-ProBook-640-G1:~/Bureau$
```

III. Avantages, inconvénients et limite de la programmation en OpenMP

1. Avantages

- Simple à implémenter ;
- Les communications sont à la charge du compilateur (communication implicites). Ce qui rend son utilisation beaucoup plus facile ;
- Lors de la parallélisation, seules les régions parallèles sont spécifiées, les régions séquentielles restent inchangées ;
- Un même programme peut contenir plusieurs régions parallèles et séquentielles.

2. Inconvénients et limite de la programmation en OpenMP

- Problème de localité des données ;
- Mémoire partagée mais non hiérarchique ;
- Les surcouts dus u partage du travail et à la création/gestion des threads peuvent se révéler importants, particulièrement lorsque la granularité du code parallèle est petite/faible (boules de petites tailles de tableaux).

Passage à l'échelle limité (efficacité sur plusieurs cœurs), parallélisme modéré : dans la pratique, on observe généralement une extensibilité limitée de codes même lorsqu'ils sont bien optimisés.