Cours de Calcul Parallèle

Master I Informatique

Spécialité : Intelligence Artificielle et Multimédia

Pr. Rachid Seghir
r.seghir@univ-batna2.dz

- **Objectif du cours :** l'objectif du cours est de permettre aux apprenants d'acquérir les connaissances nécessaires pour le développement de programmes parallèles qui exploitent le mieux possible la puissance des machines parallèles modernes. En particulier les machines parallèle à mémoire partagée et les machines parallèle à mémoire distribuée.

Références :

Livre de référence : Thomas Rauber and Gudula Rünger, Parallel Programming for Multicore and Cluster Systems, Springer 2013.

- **Manuels des bibliothèques de programmation parallèle :**

OpenMP reference guide : https://www.openmp.org/resources/refguides/

Open MPI documentation : https://www.open-mpi.org/doc/

Plan du cours

- Introduction et concepts de base
- Architectures parallèles
- Modèles de programmation parallèles
- Analyse de performances de programmes parallèles
- Langages et environnement parallèles
- Applications

Introduction et concepts de base

Utilisation du parallélisme

Utilisation classique du parallélisme : la programmation parallèle est largement utilisée en calcul scientifique à haute-performances.

- Les simulations (précises ou de problèmes complexes) nécessite de plus en plus de puissance de calcul et d'espace mémoire.

- Exemples de simulations :
  - ✓ Prévisions météorologiques (développement futur de l'atmosphère à court et long terme).
  - ✓ Conception d'avions, aérodynamisme, automobile, sous-marins furtifs.
  - ✓ Simulations de crash en industrie automobile (résultats plus précis à moindre coût). etc.

- Il s'agit de simulations basées sur des modèles mathématiques complexes

Utilisation du parallélisme (2)

Modèles mathématiques de simulations : dans le cas des simulations sub-mentionnées, il s'agit souvent de modèles de calcul numérique :

- Valeurs et vecteurs propres.
- Méthodes des éléments finis.
- Equations aux dérivées partielles.
- Calcul matriciel (la multiplication en particulier).

Autres utilisations du calcul parallèle :

- Vision par ordinateur.
- Reconnaissance de parole.
- Application graphique en industrie de cinéma et de publicité.
- etc.

Calcul parallèle et Intelligence Artificielle

- L'**intelligence artificielle** est une discipline scientifique recherchant des méthodes de résolution de problèmes à forte complexité logique ou algorithmique.

- Elle désigne également, les dispositifs imitant ou remplaçant l'humain dans certaines mises en œuvre de ses fonctions cognitives

- Les exemples types des méthodes de résolutions issues de l'intelligence artificielles sont celles inespérées de la nature :Algorithmes génétiques, etc.

- Ce types de méthodes de résolution sont de nature complexes, que ce soit en temps de de calcul ou en capacité de stockage

- Le **besoin du calcul parallèle** est ainsi évident pour les applications de l'intelligence artérielle.

Calcul parallèle et SRI

- Le principal rôle d'un système de recherche d'information est de retrouver un ensemble de documents pertinent à une requête utilisateur.
- La recherche se fait bien évidemment dans une base de documents (corpus).
- La phase de recherche est souvent précédé d'une phase d'indexation des documents du corpus afin d'en extraire une représentation compacte et plus propice aux algorithmes de recherche.
- Les temps d'indexation et de recherche sont proportionnels à la taille du corpus.
- Pour les corpus gigantesques, le **besoin du calcul parallèle** est évident, que ce soit pour la phase indexation ou recherche

Plateformes de calcul parallèle

Le calcul parallèle ne peut avoir lieu que sur des **plateformes de calcul Parallèle.**

Parmi les plateformes de calcul parallèle actuelles, on distingue :

- **Les machines multi-cours** :
  - ✓ Processeurs possédant plusieurs cœurs physiques (unités de calcul) gravés au sein de la même puce.
  - ✓ Largement disponibles à des coûts de plus en plus faibles.
  - ✓ De nos jours, le nombre de cœurs d'un processeur type pourrait atteindre des centaines (**Intel® Xeon Phi™7290** 16GB, 1.50 GHz, 72 cœurs)

Plateformes de calcul parallèle (2)

- **Les supercalculateurs** :
  - ✓ Ordinateurs conçus pour atteindre les plus hautes performances possibles avec les technologies disponibles au moment de leur conception
  - ✓ Le projet Top 500 fait une classification des 500 supercalculateurs les plus performants dans le monde.
  - ✓ Ces machines ne sont généralement pas à la portée de tout le monde.

- **Autres plateformes de calcul parallèle :** peuvent être élaborées en faisant collaborer des ordinateurs ordinaires interconnectés, telles que :
  - ✓ **Les clusters** (ensemble d'ordinateurs reliées au sein d'un réseau local).
  - ✓ **Les grilles de calcul** qui sont des agrégations de ressources de calcul hétérogènes et distribuées sur des sites distants.
  - ✓ **Cloud Computing**
  - ✓ **Les Processeurs graphiques (GPGPU) :** les processeurs graphiques modernes contiennent des milliers de cœurs (exp : 5120 cœurs dans la Tesla V100)
  - ✓

Conception d'algorithmes parallèles

- Pour pouvoir exploiter les plateformes de calcul parallèle actuelles, le traitement (calcul) à exécuter doit être partitionné en plusieurs parties qui seront affectées aux différentes ressources parallèles.

- Les différentes parties d'un calcul doivent être indépendantes chacune de l'autre.

- L'algorithme implémenté doit donc fournir assez de calculs indépendants pour qu'il soit convenable à une exécution parallèle.

- **Problème :** comment concevoir un tel algorithme ? et comment savoir si deux parties d'un calcul sont indépendantes ou non ?!!

- **Solution :** repenser en parallèle et utiliser des techniques de parallélisation (analyse de dépendances).

Conception d'algorithmes parallèles (2)

- La première étape de la programmation parallèle est la conception d'un algorithme ou programme parallèle pour une application donnée.

- La conception commence par la décomposition des calculs de l'application en plusieurs parties dites **tâches**.

- La décomposition d'une applications en tâches peut être compliquée et difficile (il y a différentes possibilités de décomposition pour une même application).

  Par exemple, selon la **granularité** (taille de tâches en termes du nombre d'instructions).

- Mais le **parallélisme potentiel** est une propriété intrinsèque à une application, ce qui influe sur la façon de décomposer une application.

  => La décomposition d'une application en tâches est l'une des principales activités intellectuelles dans le développement de programme parallèles (difficile à **automatiser)**

Ordonnancement de tâches et affectation

- Les tâches d'une application sont programmées dans un langage ou environnement parallèle et sont assignées à des **processus** ou **threads**.

- Les processus ou treads sont ensuite assignés à des unités de calcul physiques pour être exécutés.

- L'assignation des tâches à des processus ou threads est dite **ordonnancement** (*scheduling*). Il détermine l'ordre dans lequel les tâches sont exécutées.

- L'ordonnancement peut se faire à la main dans le code source ou par l'environnement de programmation au moment de la compilation ou dynamiquement au moment de l'exécution

- L'assignation de processus ou threads à des unités de calcul (processeurs ou cœurs) est dite **affectation** (*mapping*).

Dépendances entre tâches

- L'affectation est généralement faite par le système, mais elle peut être influencées par le programmeurs dans certains cas.

- Les tâches d'une application peuvent être indépendantes, mais elles peuvent aussi être dépendantes les unes des autres. On distingue :

- ✓ **La dépendance de données** lorsqu'une tâche a besoin d'une donnée produite par une autre.
- ✓ **La dépendance de contrôle** lorsque l'exécution d'une tâche dépend du résultat de l'exécution d'une autre.

- Les dépendances entre les tâches sont donc des contraintes pour l'ordonnancement.

- Les programmes parallèles ont besoin de **synchronisation** et de **coordination** de processus et threads pour s'exécuter correctement

Organisation mémoire

- Les méthodes de synchronisation et de coordination en calcul parallèle sont fortement liées à la façon dont les informations sont échangées entre les processus ou threads.
- La façon dont les informations sont échangées dépend de l'organisation de la mémoire du hardware.
- Une classification grossière de l'organisation mémoire distingue entre les machines à **mémoire partagée** et celles à **mémoire distribuée.**
- Souvent, le terme *thread* est lié à la mémoire partagée et le terme *processus* est lié à la mémoire distribuée.

Mémoire partagée vs Mémoire distribuée

Machine à mémoire partagée	**Machine à mémoire distribuée**
✓ Les données sont stockées dans une même **mémoire partagée** globale.	✓ Chaque processeur dispose d'une **mémoire privée**.
✓ Les données peuvent être accédées par tous les processeurs ou cœurs de processeur.	✓ Une mémoire privée d'un proc. ne peut pas être accédées par les autres proc.
✓ L'échange d'information entre threads se fait moyennant des **variables partagées** écrites par un thread et lues par un autre tread.	✓ L'échange d'information se fait par l'envoie de données d'un processeur à un autre via un réseau d'interconnexion (moyennant des **opérations de communication**)
✓ Le comportement correct du programme global est assuré par la **synchronisation** des threads (coordination des accès aux données partagées)	✓ Pas besoin de synchroniser les accès mémoire.

Opérations de barrière

- Des **opérations de barrière** spécifiques constituent une autre forme de synchronisation applicables aux machines à mémoire partagée et celle à mémoire distribuée.

- Dans ce cas, tous les processus ou threads doivent attendre à une barrière de synchronisation jusqu'à ce que tous les autres processus ou threads atteignent cette barrière.

- Les processus ou threads continuent à exécuter le code situé après la barrière seulement s'il ont tous terminé d'exécuter le code situé avant la barrière.

Temps d'exécution parallèle

- Un autre aspect important du calcul parallèle est le **temps d'exécution parallèle** (temps de calcul sur processeurs ou cœurs + temps de d'échange de données ou de synchronisation).

- Le temps d'exécution parallèle d'une application est le temps écoulé entre le début de l'exécution de l'application sur le premier proc. et la fin de son exécution sur tous les proc. (il doit être inférieur au temps séquentiel sur un seul proc.).

- Le temps d'exécution parallèle est influencé par :

✓ la distribution du travail sur les processeurs ou cœurs.
✓ le temps d'échange d'information ou de synchronisation.
✓ le **temps d'inactivité** (*idle time*) pendant lequel le processeur ne peut faire rien d'utile mais attend un certain événement.

Temps d'exécution parallèle (2)

- En général, le temps d'exécution parallèle est plus petit lorsque :
  - ✓ la charge du travail est assignée équitablement aux différents processeurs ou cœurs, i.e., **équilibrage de charge** (lood balancing).
  - ✓ Les temps d'échange d'information, de synchronisation et d'inactivité sont petits.
- Trouver une stratégie d'ordonnancement et d'affectation qui mène à un bon équilibrage de charge et moins de temps d'échange d'information, de synchronisation et d'inactivité est souvent une tâche difficile (compromis).

Accélération et efficience

- Pour une évaluation quantitative du temps d'exécution de programmes parallèles, on utilise des mesures de coût, telles que l'**accélération** (*speedup*) et l'**efficience** (*effeciency*).

- L'accélération est définie par la formule suivante :

$$S_p = \frac{T_1}{T_p}$$

  - ✓ p est le nombre de processeurs
  - ✓ T_1 est le temps d'exécution de l'algorithme séquentiel
  - ✓ T_p est le temps d'exécution de l'algorithme parallèle sur p processeurs

- On dit que l'accélération est **linéaire** ou **idéale** lorsque Sp = p.

- L'accélération est dite **absolue** si T_1 est le meilleure temps d'exécution de l'algorithme séquentiel.

Accélération et efficience (2)

- L'accélération est dite **relative** si T_1 est le temps d'exécution du même algorithme parallèle sur un seul processeur (c'est le cas par défaut).

- L'efficience est une mesure de performances définie par :

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \times T_p}$$

- Typiquement, c'est une valeur entre 0 et 1 qui estime combien les processeurs sont bien utilisés pour la résolution du problème, comparé à combien de temps est perdu dans la communication et la synchronisation.

- Les algorithmes s'exécutant avec une accélération linéaire ou sur un seul processeur ont une efficacité égale à 1.

Accélération et efficience (3)

- Les algorithmes difficiles à paralléliser ont une efficience telle que :

$$\frac{1}{\log p}$$

- Dans ce cas, plus le nombre de processeurs est grand plus l'efficience est proche de zéro.

- **NB.** Dans certains cas (rares), on peut avoir un accélération supérieure à p en utilisant p processeurs. Une telle accélération est dite **super linéaire.**

- **Question :** quelle peut être la raison d'une accélération super linéaire ?!!

Les architectures parallèles

Classification des machines parallèles

- Les machines parallèles (MP) ont été utilisées depuis plusieurs années, et différentes alternatives architecturales ont été proposées.

- En général, une **machine parallèle** peut être vue comme **un ensemble d'éléments de calcul qui peuvent communiquer et coopérer pour résoudre un grand problème plus rapidement**.

- Cette définition est assez générale et ne prend pas en compte certains détails, tels que :

  - ✓ le nombre et la complexité des éléments de calcul,
  - ✓ la structure du réseau d'interconnexion entre les éléments de calcul.
  - ✓ la coordination du travail entre les éléments de calcul, ainsi que certaines caractéristiques importantes du problème à résoudre.

- Pour une investigation plus détaillée, il est utile de faire une classification par rapport aux caractéristiques importantes des MP.

Classification de Flynn

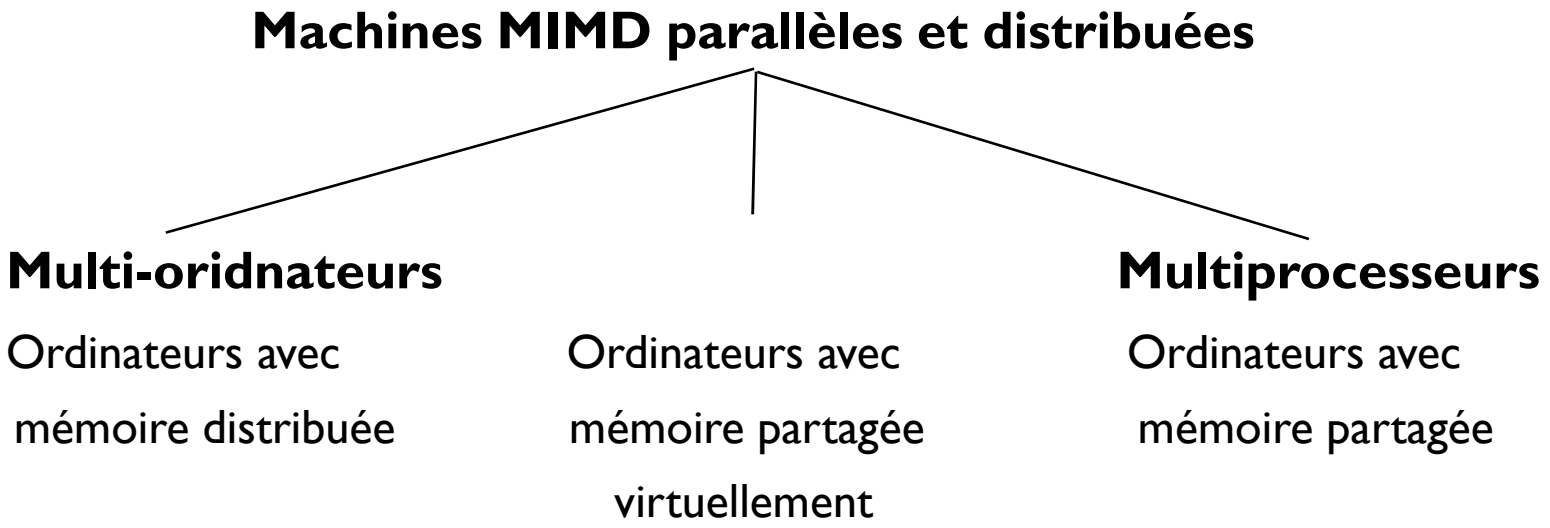
- Un modèle simple de classification de MP et celui définie par la taxonomie de **Flynn**.

- Les machines sont classées selon deux flots : le **flot d'instructions** et le **flot de données**. Ces flots peuvent être simples ou multiples.

		Flot de données	
		Single	Multiple
flot d'instructions	Single	SISD (Von Neumann)	SIMD (Vectorielles ou Cellulaires)
	Multiple	MISD (Pipeline)	MIMD (Multi-processeurs et Multi-ordinateurs)

Classification selon l'organisation mémoire

- Pratiquement, toutes les machines parallèles actuelles sont basées sur le modèle MIMD.
- Une autre classification des machines MIMD peut être faite selon leur organisation mémoire.



Classification selon l'organisation mémoire (2)

- Du point de vue programmeur, on peut distinguer entre les machines avec **espace d'adressage distribué** et celle avec **espace d'adressage partagé**.

- Ce point de vue ne doit pas être nécessairement conforme à la mémoire physique.

- Une machine parallèle avec une mémoire physiquement partagée peut être vue par le programmeur comme une machine avec un espace d'adressage partagée lorsque l'environnement de programmation correspondant est utilisé.

- Les machines avec mémoire physiquement distribuée (resp. partagée) sont connues par le nom de **DMMs** « *Distrubitet Memory Machines* » (resp. **SMMs** « *Shared Memory Machines* »).

DMMs et modèle de passage de messages

- Les DMM sont généralement liées au modèle de programmation par passage de message.

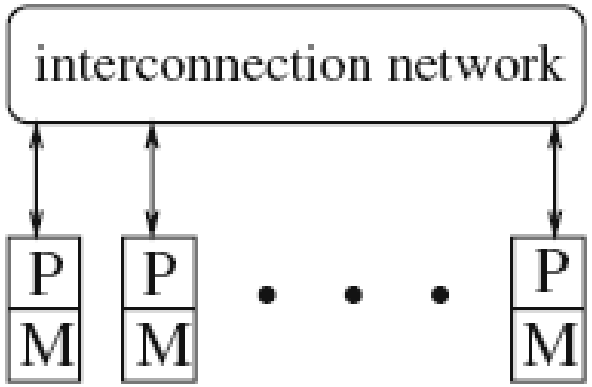
- Pour exécuter un passage de message, deux processeur P_A et P_B sur différents nœuds A et B exécutent des opérations d'envoi et de réception.

- Quand P_B a besoin de données se situant sur le noeud A.
  - ✓ P_A exécute une **opération d'envoi** (*send*) contenant les données pour le processeur destinataire P_B.
  - ✓ P_B exécute une **opération de réception** (receive) spécifiant le buffer de réception pour le stockage des données envoyées par le processeur source P_A.

- Plusieurs architectures de DMM ont été proposées et expérimentées. En particulier du point de vue **réseau d'interconnexion.**

Organisation des DMMs

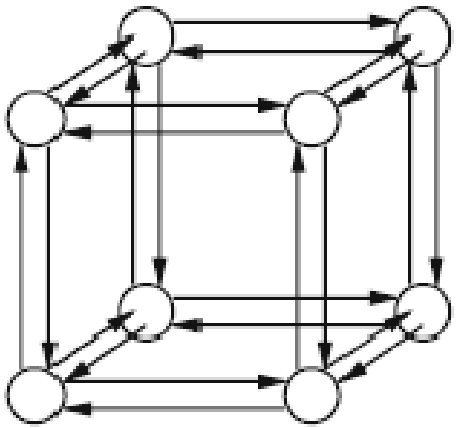
a)



P = processor

M = local memory

b)

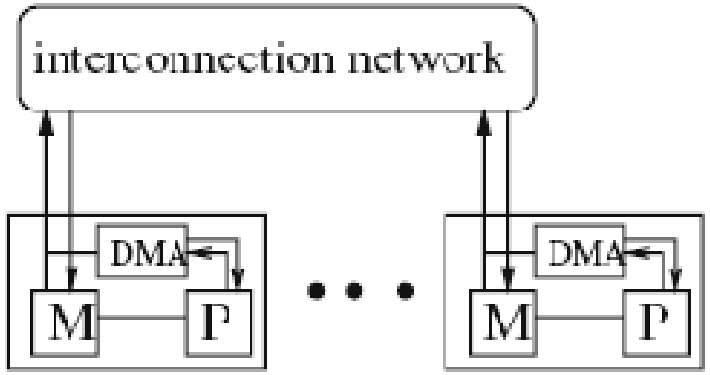


node consisting of processor and local memory

computer with distributed memory with a hypercube as interconnection network

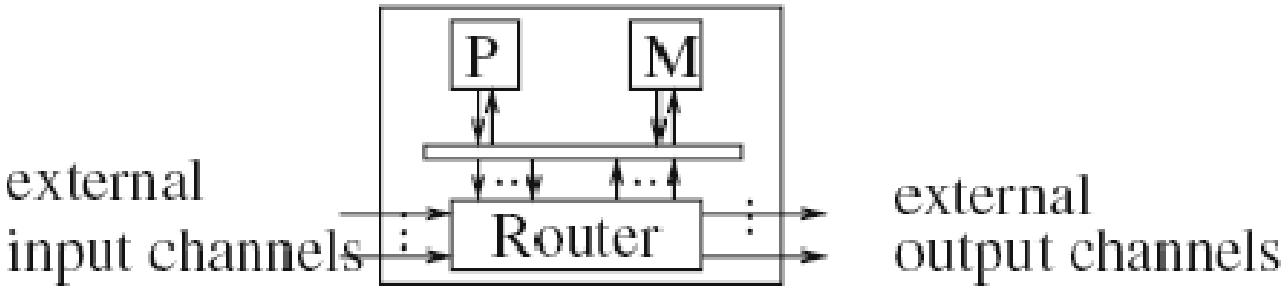
Organisation des DMMs (2)

c)



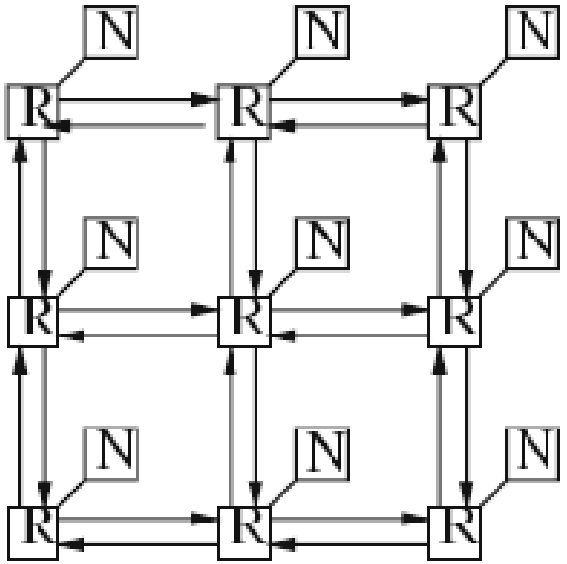
DMA (direct memory access)
with DMA connections
to the network

d)



Organisation des DMM (3)

e)



R = Router

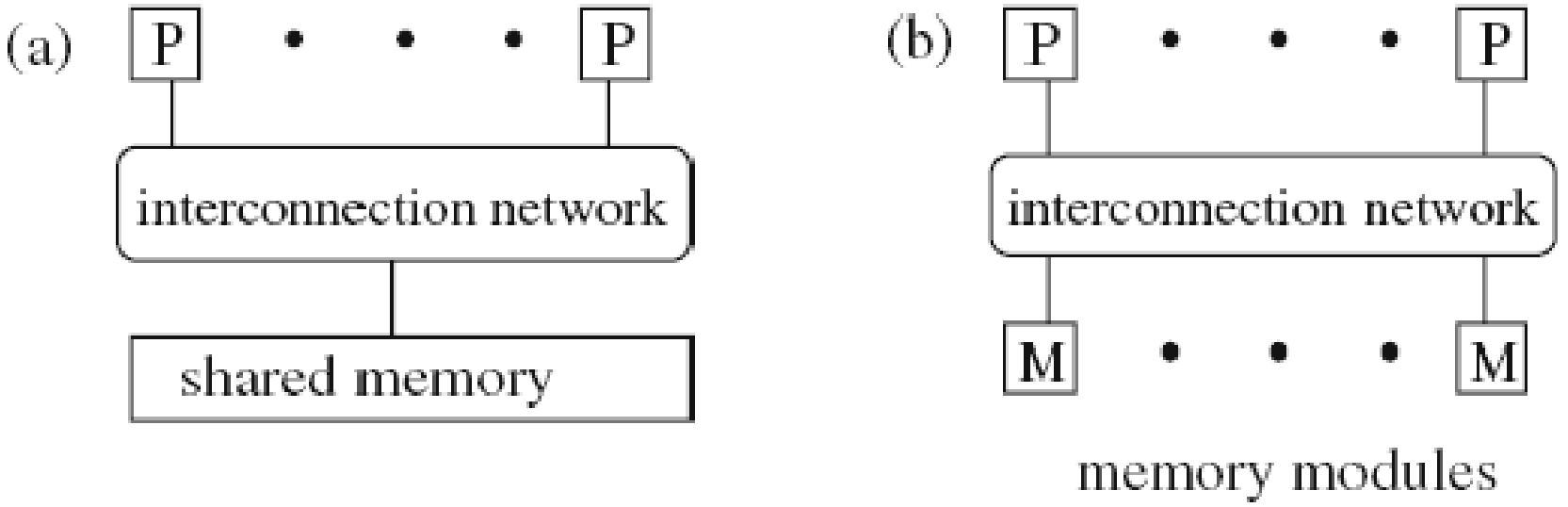
N = node consisting of processor and local memory

Réseau d'interconnexion sous forme de maille pour connecter les routeurs des nœuds (processeur-mémoire).

SMM et modèle de variables partagées

- Les SMMs consistent en un nombre de processeurs ou cœurs, à mémoire physiquement partagées (mémoire globale), et un réseau d'interconnexion pour connecter les processeurs (ou coeurs) à la mémoire.

- La mémoire partagée peut être implémentée comme un ensemble de **modules de mémoire** séparés fournissant un espace d'adressage commun.



SMM et modèle de variables partagées (2)

- L'échange d'information entre les processeurs se fait via la mémoire globale par **lecture et écriture de variables partagées**.
- Le modèle de programmation pour les SMMs est donc l'utilisation de variables partagées qui peuvent être accédées par tous les processeurs.
- Mais l'accès concurrent aux variables partagées par plusieurs proc. peut causer le problème de « **concurrence critique** » => besoin de mécanisme de gestion du pb de concurrence.
- La communication via des variables partagées est facile puisqu'il n'y a pas besoin de réplication de données comme c'est parfois le cas des DMMs.
- Mais techniquement, la réalisation de SMMs a besoin de plus d'effort. En particulier à cause du réseau d'interconnexion qui doit fournir un accès rapide à la mémoire globale pour chaque processeur.

Les MultiProcesseurs Symétriques –SMPs-

- Une variante des SMMs, dite **SMP** (Symetric Multiprocessor), est une SMM ayant une seule mémoire partagée avec un temps d'accès uniforme (**UMA** *Uniforme Memory Access*) à partir de tous les processeurs vers n'importe quelle location mémoire (cas des processeurs multicoeurs).

- Les SMPs ont souvent un petit nombre de processeurs connectés via un **bus central** fournissant également l'accès à la mémoire partagée (le petit nombre de processeurs est justifié par la bande passante limitée du bus central).

- Les SMPs peuvent être utilisés comme des nœuds d'une machine parallèle plus grande, en employant un réseau d'interconnexion pour l'échange de données entre les processeurs des différents SMPs.

- Le systèmes résultants, appelés **DSMs** (Distributed Shared Memory), ont un **temps d'accès non uniforme à la mémoire globale**, d'où ils sont aussi appelés **NUMA** (Non-Uniform Memory Access).

Réduction du temps d'accès à la mémoire

- Le temps d'accès à la mémoire a une grande influence sur les performances d'un programme, et ce même pour des machines à espace d'adressage partagé.

- Si la capacité des puces DRAM utilisées en mémoire centrale augmente par environ 60% par an, le temps d'accès à ces dernières n'augmente que par environ 25% par an !

- La technologie de la mémoire ne peut suivre celle du processeur dans son essor. En effet, l'écart entre la vitesse du processeur et celle de temps d'accès à la mémoire progresse d'environ 50% par an !

- L'organisation adéquate des accès à la mémoire est devenue de plus en plus importante pour améliorer les performances des programmes (séquentiels ou parallèle), en particulier lorsqu'un espace d'adressage partagé est utilisé.

Réduction du temps d'accès à la mémoire

- La réduction de la latence moyenne observée par un processeur lorsqu'il accède à la mémoire peut améliorer significativement les performances résultantes d'un programme.

- Deux principales approches ont été développées pour réduire la latence moyenne des accès mémoire :

  - ✓ La simulation de **processeurs virtuels** par chaque processeur physique (Multithreading).
  - ✓ L'utilisation de **mémoires caches locales** pour stocker les valeurs des données qui sont accédées souvent.

Le Multithreading

- L'idée du **Multithreading** (multithreading entrelacé) est de cacher la latence des accès mémoire en simulant un certain nombre de processeurs virtuels pur chaque processeur physique.

- Le processeur physique contient un conpteur ordinal séparé ainsi q'un ensemble séparé de registres pour chaque processeur virtuel.

- Après l'exécution d'une instruction machine (par un processeur virtuel), un switch implicite est effectué vers le processeur virtuel suivant.

- Les processeurs virtuels sont simulés par le processeur physique à tour de rôle (tourniquet ou *round-robin*).

- Le nombre de processeurs virtuels par processeur physique doit être choisi de façon à ce que le temps compris entre les exécutions de deux instructions successives d'un processeur virtuel soit suffisant pour le chargement des données requises à partir de la mémoire.

Le Multithreading (2)

- La latence mémoire est donc cachée par l'exécution des instructions des autres processeurs virtuels.

- On distingue généralement entre deux types de multithreading :
  - ✓ Le **multithreading à grain fin** : lorsqu'un switch entre processeurs virtuels s'exécute après chaque instruction.
  - ✓ Le **multithreading à gros gain** : lorsque le switch entre processeurs virtuels s'exécute seulement lors de situations coûteuses (comme les défauts de caches de niveau 2).
- Il y a une autre forme de multithreading dite **multithreading simultané** (**Hyperthreading**) dans laquelle des instructions de différents threads sont ordonnancées pour s'exécuter dans le même cycle, si nécessaire, affin d'utiliser les unité fonctionnelles du processeur plus efficacement,

- C-à-d, les instructions des différents threads sont en concurrence sur les unités fonctionnelles du processeur.

Les mémoires caches

- Une **mémoire cache** ou **antémémoire** est une mémoire qui enregistre temporairement des copies de données provenant d'une autre source de donnée comme les mémoires.

- Le rôle de la mémoire cache est de diminuer le temps d'accès (en lecture ou en écriture) d'un matériel informatique (en général, un processeur) à ces données.

- La mémoire cache est plus rapide et plus proche du matériel informatique qui demande la donnée, mais plus petite que la mémoire pour laquelle elle sert d'intermédiaire.

- La mémoire cache est généralement utilisée pour stocker des copies des données qui sont souvent utilisées par le processeur afin d'éviter (lorsque c'est possible) les accès (coûteux) à la mémoire centrale.

Principe de fonctionnement des M. caches

- Une fois les données sont stockées dans le cache, l'utilisation future de ces données peut être réalisée en accédant à la copie en cache plutôt qu'en récupérant ou recalculant les données, ce qui abaisse le temps d'accès moyen.

- Le principe de fonctionnement des mémoires caches est comme suit :

1. l'élément demandeur (processeur) demande une information ;
2. le cache vérifie s'il possède cette information. S'il la possède, il la retransmet à l'élément demandeur – on parle alors de **succès de cache** (***cache hit***). S'il ne la possède pas, il la demande à l'élément fournisseur (mémoire principale par exemple) – on parle alors de **défaut de cache** (***cache miss***);
3. l'élément fournisseur traite la demande et renvoie la réponse au cache ;
4. le cache la stocke pour utilisation ultérieure et la retransmet à l'élément demandeur au besoin.

Localités spatiale et temporelle

- Si les mémoires cache permettent d'accroître les performances, c'est en partie grâce à deux principes qui ont été découverts suite à des études sur le comportement des programmes informatiques :
  - ✓ Le principe de **localité spatiale** qui indique que l'accès à une donnée située à une adresse X va probablement être suivi d'un accès à une zone toute proche de X ;
  - ✓ Le principe de **localité temporelle** qui indique que l'accès à une zone mémoire à un instant donné a de fortes chances de se reproduire dans la suite immédiate du programme.
- Plus ces deux principes sont assurés, moins la mémoire centrale est accédée.
- Plusieurs méthodes de **transformation automatique de programmes** se focalisent sur l'amélioration de la localité spatiale et/ou temporelle des programme affin de les optimiser (améliorer leurs performances).

Gestion de la mémoire cache

- La gestion de la mémoire cache est assurée par le hardware en employant une stratégie dite associative par ensemble (*set-associative*).

- La mémoire cache ne peut pas contenir toute les données de la mémoire principale, il faut définir une méthode indiquant à quelle adresse de la mémoire cache doit être écrite une **ligne** de la mémoire principale. Cette méthode s'appelle le **mapping** (correspondance).

- Il existe trois types de mapping répandus dans les caches aujourd'hui :
  - les mémoires caches **complètement associatives** (***fully associative cache***) ;
  - les mémoires caches **N-associatives** (***N-way set associative cache***) ;
  - les mémoires caches **directes** (*direct mapped cache*).

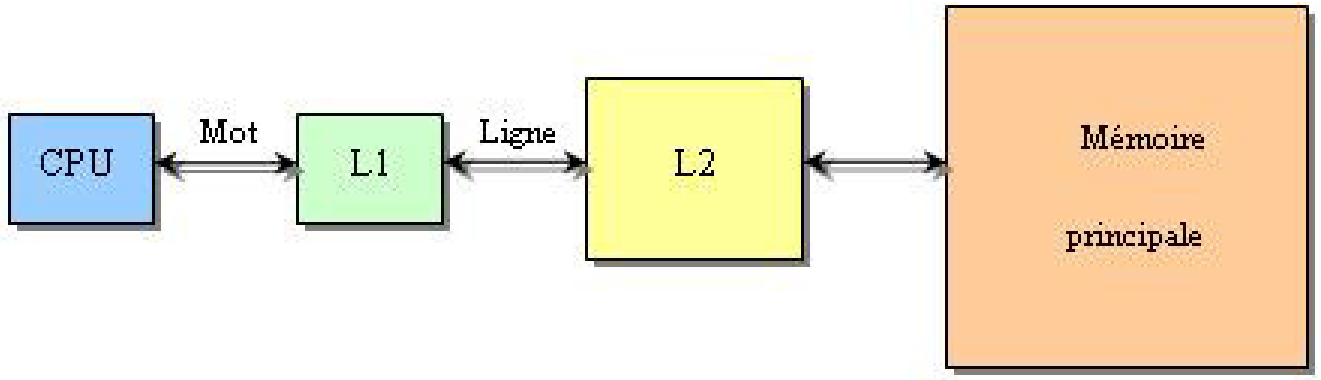
- ***N.B. ligne** = plus petit élément de données qui peut être transféré entre la mémoire cache et la mémoire de niveau supérieur. **mot** = plus petit élément de données qui peut être transféré entre le processeur et la mémoire cache.*

Les niveaux de caches

- Plusieurs niveaux de caches sont généralement placés entre le processeur et la MC, à partir du niveau L1 (petit, très rapide et chère) jusqu'à plusieurs niveaux (L2, L2) de tailles plus grandes mais avec des temps d'accès plus grands.

- Pour un processeur type, le temps d'accès au niveau de cache 1 peut prendre 2-4 cycles, tendis que l'accès à la mémoire centrale peut prendre des centaines de cycles !

- Ces niveaux de caches peuvent être situés dans ou hors du microprocesseur



Algo. de remplacement de lignes de cache

- Les caches associatifs de N voies et complètement associatifs impliquent le mapping de différentes lignes de la mémoire de niveau supérieur sur le même set (ensemble).

- Lorsque le set de lignes de la mémoire cache, où une ligne de la mémoire supérieure peut être mappée, est rempli, il faut désigner la ligne qui sera effacée au profit de la ligne nouvellement écrite.

- Le but des algorithmes de remplacement des lignes de cache est de choisir cette ligne de manière optimale. Ces algorithmes doivent être implémentés en hardware pour les mémoires caches de bas niveau afin d'être les plus rapides possible et de ne pas ralentir le processeur. Cependant, ils peuvent être implémentés en software pour des caches de niveau supérieur.

- La majorité des algorithmes reposent sur le principe de localité pour tenter de prévoir le futur à partir du passé.¨Parmi ces algorithme on trouve : **FIFO** (*First In First Out*), **LRU** (Least Recently Used), etc.

Mémoires caches et architectures parallèles

- Les mémoires caches sont utilisées dans les machines mono-processeur, mais elles jouent également un rôle important pour les SMPs et les machines parallèles avec les différents type d'organisation mémoire.

- Les SMPs utilisent une mémoire partagée. Lorsqu'une donnée partagée est accédée par plusieurs processeurs, elle peut être dupliquée dans des caches affin de réduire la latence des accès.

- Chaque processeur doit avoir une **vue cohérente** du système de la mémoire, c-à-d, chaque accès en lecture doit retourner la valeur la plus récemment écrite quelque soit le processeur ayant effectué cette écriture.

- Lorsqu'un processeur écrit (modifie) une donnée partagée dans son cache, une mise à jour de cette donnée doit se faire dans la MC mais aussi dans les caches des processeurs qui contiennent une copie de la même donnée.

- La cohérence du cache est assuré par un protocole dit : **protocole de cohérence de cache** (*cache coherency protocol*).

Modèles de programmation parallèle

Modèle de programmation parallèle

- Le modèle de programmation parallèle décrit un système de calcul parallèle en termes des sémantiques du langage ou de l'environnement de programmation.

- Il spécifie la façon dont une machine parallèle est vue par un programmeur, en décrivant comment ce dernier peut coder (implémenter) un algorithme.

- Cette vue est influencée par la conception architecturale, le langage de programmation, le compilateur, ou les bibliothèques dynamiques (*runtime libraries*).

- Il peut donc exister différents modèles de programmation parallèle pour une même architecture.

Modèle de programmation parallèle (2)

- Les modèles de programmation parallèles peuvent se différer par :
  - ✓ Le niveau de parallélisme exploité dans l'exécution parallèle (instruction, statement, procédure, boucles parallèles.)
  - ✓ Spécification implicite ou explicite (par l'utilisateur) du parallélisme.
  - ✓ La façon dont les parties parallèles sont spécifiées.
  - ✓ Le mode d'exécution des unités parallèles (SIMD ou SPMD, …)
  - ✓ La façons dont les informations sont échangées entre les unités parallèles (communication explicite ou variables partagées).
  - ✓ Etc.
- Dans ce cours, nous nous intéressons aux modèles de programmation parallèle basés sur les concepts de processus et threads (espace d'adressage distribué ou partagé).

Parallélisation de programmes

- **La parallélisation** d'un programme ou d'un algorithme donné est typiquement effectué sur la base du modèle de programmation parallèle utilisé.
- Indépendamment du modèle de programmation spécifique, on peut identifier des étapes types pour effectuer la parallélisation.
- Pour transformer des calculs séquentiels en un programme parallèle, leurs dépendances de contrôle et de données doivent être pris en considération pour assurer que le programme parallèle produit les mêmes résultats que le programme séquentiel pour toutes les valeurs possibles des entrées.
- Pour effectuer la parallélisation d'une manière systématique, elle peut être partitionnée en plusieurs étapes.
  - ✓ Décomposition des calculs en tâches.
  - ✓ Assignation des tâches aux processus ou threads.
  - ✓ Affectation des processus ou threads aux processeurs physiques ou cœurs de processeur.

Décomposition des calculs en tâches

- Les calculs séquentiels sont décomposés en tâches, et les dépendances entre ces tâches sont déterminées.

- Les tâches sont les plus petites unités du parallélisme, elle peuvent être identifiées à différents niveaux d'exécution (niveau instruction, parallélisme de donnée, parallélisme fonctionnel).

- Selon le modèle de la mémoire utilisé, une tâche peut accéder à l'espace d'adressage partagé ou exécuter des opérations de passage de message.

- La décomposition en tâches peut s'effectuer de manière statique (phase d'initialisation au début du programme) ou de manière dynamique (durant l'exécution du programme).

- Le rôle de la décomposition est de générer suffisamment de tâches pour rendre tous les processeurs ou cœurs occupés à tout moment de l'exécution du programme.

Décomposition des calculs en tâches (2)

- D'un autre côté, les tâches doivent contenir assez de calculs de façon à ce que le temps d'exécution de la tâche soit grand par rapport au temps requis par son ordonnancement et affectation.

- La **granularité** des tâche se mesure par le temps (ou le nombre) de leurs calculs.

- Les tâches ayant plusieurs calculs ont une **granularité à gros grain** (*coarse-grained granularity*).

- Les tâches ayant un petit nombre de calculs ont une **granularité à grain fin** (*fine-grained granularity*).

- La phase de décomposition doit trouver un bon compromis entre le nombre de tâches et leur granularité (exploiter efficacement les processeurs ou cœurs avec moins de temps d'ordonnancement et d'affectation).

Assignation de tâches

- Un processus ou thread représente un flot de contrôle exécuté par un processeur ou cœur physique.
- Un processus ou thread peut exécuter différentes tâches l'une après l'autre
- Le nombre de processus ou threads ne doit pas être nécessairement le même que le nombre de processeurs ou cœurs physiques, mais souvent on utilise le même nombre.
- Le but principal de la **phase d'assignation** est d'assigner les tâches de telle façons à ce qu'un bon **équilibrage de charge** est atteint. C-à-d, chaque processus ou thread doit avoir à peu près le même nombre de calculs à effectuer,
- Mais le nombre des accès à la mémoire (espace d'@ partagé) ou le nombre d'opérations de communication (espace d'@ distribué) doit aussi être pris en considération.

Assignation de tâches (2)

- Dans le cas de l'espace d'adressage partagé, il est utile d'assigner deux tâches travaillant sur la même donnée au même thread, puisque ceci mène à une bonne utilisation du cache.

- L'assignation de tâches aux processus ou threads est aussi dite **ordonnancement** (*scheduling*) .

- L'ordonnancement peut s'effectuer d'une manière statique (phase d'initialisation au démarrage du programme), comme elle peut s'effectuer de manière dynamique (durant l'exécution du programme).

- Un **algorithme d'ordonnancement** permet de définir, pour chaque tâche, le temps de démarrage et l'unité d'exécution, tel que les **contraintes de précédence et de capacité** sont satisfaites, et tel qu'une fonction objective donnée (le temps globale d'exécution) est optimisé .

- Les algorithmes d'ordonnancement sont généralement **NP-complet**.

Affectation de processus ou threads

- La phase d'**affectation** (*mapping*) consiste en l'assignation de processus ou threads aux processeurs ou cœurs de processeur physiques sur lesquels ils seront exécutés.

- Dans le cas le plus simple, chaque processus ou thread est affecté à un processeur ou cœur séparé (unité d'exécution).

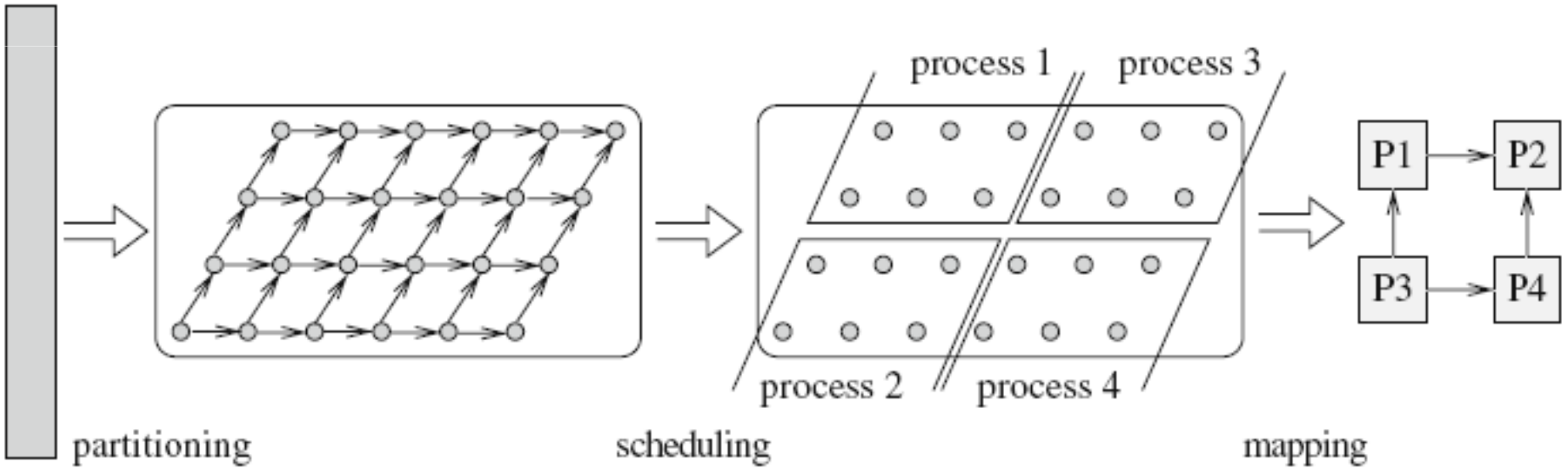
- Si le nombre de threads est supérieur au nombre de cœurs de processeur, plusieurs threads doivent être affectés à un seul cœur.

- Cette affectation peut être effectué par le système d'exploitation, mais elle peut aussi être supportée par les instructions du programme.

- Le but principal de la phase d'affectation est d'avoir une utilisation équitable des processeurs ou cœurs de processeur, tout en maintenant une communication minimale entre les processeurs.

Etapes de parallélisation d'un programme

- Illustration des étapes de parallélisation d'un algorithme séquentiel donné : L'algorithme est d'abord partitionné en tâches, et les dépendances entre ces tâches sont identifiées. Les tâches sont ensuite assignées aux processus par un ordonnanceur. Enfin, les processus sont affecté aux processeurs physiques P1, P2, P3 et P4 sur lesquels il seront exécutés.



Niveaux de parallélisme

- Les calculs effectués par un programme donné fournissent des opportunités pour l'exécution parallèle à différents niveaux (instruction, statement, boucle, fonction).

- Selon le niveau de parallélisme considéré, des tâches de différentes granularités résultent.
  - ✓ Au niveau instruction ou statement, des tâches à grain fin résultent lorsqu'un petit nombre d'instructions ou statements sont groupées pour former une tâche.
  - ✓ Au niveau fonction, des tâches à gros grain résultent lorsque les fonctions utilisées pour former une tâche contiennent un nombre important de calculs.
  - ✓ Au niveau boucle, des tâches à grain moyen peuvent être considérées puisqu'une boucle consiste généralement en plusieurs statements.

Parallélisme au niveau instruction

- Plusieurs instructions d'un programme peuvent être exécutées en parallèle (en même temps) si elle ne dépendent pas les unes des autres.

- L'existence de l'une de ces dépendances de données entre deux instructions I_1 et I_2 empêche leur exécution en parallèle**.**

  - **Dépendance de flot** (vraie dépendance) : il y a une dépendance de flot de I_1 vers I_2 lorsque I_1 calcule une valeur dans un registre ou une variable qui sera ensuite utilisée par I_2.
  - **Anti-dépendance** : il y a une anti-dépendance de I_1 vers I_2 lorsque I_1 utilise un registre ou une variable comme opérande qui est ensuite utilisé par I_2 pour stocker le résultat d'un calcul**.**
  - **Dépendance de sortie** : il ya une dépendance de sortie de I_1 vers I_2 si I_1 et I_2 utilisent le même registre ou la même variable pour stocker le résultat d'un calcul.

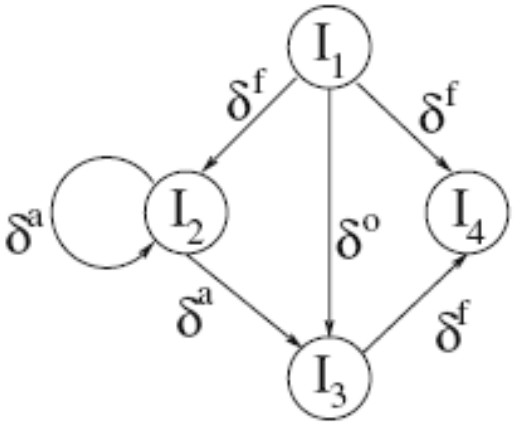
Parallélisme au niveau instruction (2)

- **Exemple :** différents types de dépendances de données entre des instructions utilisant cinq registres.

I_1: $R_1 \leftarrow R_2+R_3$	I_1: $R_1 \leftarrow R_2+R_3$	I_1: $R_1 \leftarrow R_2+R_3$
I_2: $R_5 \leftarrow R_1+R_4$	I_2: $R_2 \leftarrow R_4+R_5$	I_2: $R_1 \leftarrow R_4+R_5$
flow dependency	anti dependency	output dependency

- **Graphe de dépendances :** Les dépendances entre les instruction peuvent être représentées par un graphe orienté dans lequel les nœuds représentent les instructions et les arcs représentent les dépendances.
- Il y a un arc de I_i vers I_j s'il y a une dépendance de I_i vers I_j. Les arcs sont étiquetés par le type des dépendances qu'ils représentent.
- **Exemple :**

I_1: $R_1 \leftarrow A$
I_2: $R_2 \leftarrow R_2+R_1$
I_3: $R_1 \leftarrow R_3$
I_4: $B \leftarrow R_1$



Parallélisme de données

- Dans plusieurs programmes, la même opération est appliquée à différents éléments d'une grande structure de données (un tableau par exemple).

- Si les opérations à appliquer sont indépendantes les unes des autres, l'exécution parallèle peut être utilisée en distribuant les éléments de la structure sur les différents processeurs, et chaque processeur exécute l'opération sur l'élément qui lui est assigné (**parallélisme de données**).

- Le parallélisme de données peut être utilisé, en particulier dans le domaine du calcul scientifique.

- Pour utiliser le parallélisme de données, des langages séquentiels ont été étendus à des langages de programmation pour le parallélisme de données (***data-parallel*** *programming languages*), tels que Fortran 90/95, C*, DPCE, HPF.

- Ces langages sont similaires aux langages séquentiels mais ils contiennent des formalismes pour exprimer le parallélisme de données (**affectation de tableaux** en particulier).

Parallélisme de données (2)

- **Exemple :** affectation d'un tableau en Fortran 90

```
a(1:n)= b(0:n-1)+c(1:n)
```

est l'équivalent de :

```
for (i=1:n)
        a(i)=b(i-1)+c(i)
endfor.
```

- **Attention :** dans des langages *data-parallel* (exp. Fortran 90), l'affectation effective des éléments du tableau (partie gauche) ne se fait qu'une fois tous les accès et opérations de la partie droite sont effectués. Par exemple :

```
a(1:n)= a(0:n-1)+a(2:n+1)
```

n'est pas identique à

```
for (i=1:n)
        a(i)=a(i-1)+a(i+1)
endfor.
```

- Le parallélisme de données peut être exploité par les deux modèles de programmation parallèle (espace d'@ distribué et espace d'@ partagé).

Parallélisme de boucles

- Plusieurs algorithmes effectuent des calculs en traversant itérativement une grande structure de données.

- Le traversement itératif est généralement exprimé par une structure itérative (une **boucle**) fournie par les langages de programmation impératifs.

- Une boucle est généralement exécutée séquentiellement, c-à-d, les calculs de la $i^{ème}$ itération ne commencent que si tous les calculs des $i-1$ itérations qui la précède sont terminés. Ce schéma d'exécution est appelé **boucle séquentielle**.

- S'il n'y a pas de dépendances entre les itérations d'une boucle, ces dernières peuvent être exécutées dans n'importe qu'elle ordre, et elle peuvent aussi être exécutées en parallèle par différents processeurs. Une telle boucle est alors dite **boucle parallèle**.

Les boucles parallèles FORALL

- Le corps d'une boucle **`forall`** peut contenir une ou plusieurs affectations des éléments de tableaux.

- Si une boucle `forall` contient une seule affectation, alors son comportement est équivalent à celui de l'affectation de tableau vue précédemment, c-à-d les calculs de la partie droite de l'affectation sont d'abord effectués dans n'importe quel ordre, puis les résultats sont affectés dans n'importe quel ordre aux éléments de tableaux leur correspondant.

  Exp :
  ```
  forall (i=1:n)
      a(i)=a(i-1)+a(i+1)  ⇔  a(1:n)=a(0:n-1)+a(2:n+1)
  endforall
  ```

- Si une boucle `forall` contient plusieurs affectations, alors elles sont exécutées les une après les autres comme des affectation de tableaux, tel que l'affectation du tableau suivant ne peut commencer qu'une fois l'affectation du tableau précédent soit terminé.

Les boucles parallèles DOPAR

- Le corps d'une boucle **`dopar`** peut contenir non pas seulement des affectation de tableaux, mais aussi d'autres instructions voire d'autres boucles.

- Les itérations d'une boucle `dopar` sont exécutées en parallèle par plusieurs processeurs, chacun exécute ses itération dans n'importe quel ordre.

- Les instructions de chaque itération sont exécutées séquentiellement. En utilisant les valeurs des variables de l'état initial (avant le commencement de la boucle `dopar`). Les mises à jour de variables effectuées par une itération sont donc invisibles pour les autres itérations.

- A la fin de l'exécution de toutes les itérations, un nouvel état global est calculé en combinant les mise à jours des différentes itérations.

- Lorsque deux itérations effectuent une mise à jour d'une même variable, une seule sera prise en considération (**comportement indéterministe**)

Les boucles parallèles DOPAR (2)

- Le résultat de l'exécution des boucles `forall` et `dopar` avec le même corps de code peut être différent lorsque ce dernier continent plus d'une instruction.

- **Exemple :** résultat de l'exécution des boucles `for`, `forall` et `dopar`

```
for (i=1:4)
 a(i)=a(i)+1
 b(i)=a(i-1)+a(i+1)
endfor
```

```
forall (i=1:4)
 a(i)=a(i)+1
 b(i)=a(i-1)+a(i+1)
endforall
```

```
dopar (i=1:4)
 a(i)=a(i)+1
 b(i)=a(i-1)+a(i+1)
enddopar
```

Valeurs initiales		après la boucle `for`		après la boucle `forall`		après la boucle `dopar`	
a(0)	1						
a(1)	2	b(1)	4		5		4
a(2)	3	b(2)	7		8		6
a(4)	5	b(3)	9		10		8
a(5)	6	b(4)	11		11		10

- N.B. Une boucle `dopar` dans laquelle un élément d'un tableau calculé dans une itération n'est utilisé que dans cette dernière est dite boucle **`forall`**.

Parallélisme fonctionnel

- Plusieurs programmes séquentiels contiennent des parties indépendantes les unes des autres. Ces parties peuvent être des instructions, des blocks, des boucles ou des appels de fonction.

- Si on considère les parties indépendantes d'un programme comme des tâches, cette forme de parallélisme est appelée **parallélisme de tâches** ou **parallélisme fonctionnel.**

- Pour utiliser le parallélisme de tâches, les tâches et leurs dépendances peuvent être représentées par un **graphe de tâches** dans lequel les nœuds représentent les tâches et les arcs représentent les dépendances entre les tâches.

- Selon le modèle de programmation utilisé, une tâche peut être exécuté séquentiellement par un seul processeur, ou en parallèle par plusieurs processeurs. Dans ce dernier cas, chaque tâche peut être exécuté en exploitant le parallélisme de données (parallélisme de tâche et de données).

Parallélisme fonctionnel (2)

- Pour déterminer un plan d'exécution (ordonnancement), pour un graphe de tâches donné sur un ensemble de processeurs, un temps de démarrage doit être affecté à chaque tâche tel que les dépendances sont satisfaites.
- Un tâche ne peut commencer que si toutes les tâches dont elle dépend sont terminées.
- Le rôle d'un algorithme d'ordonnancement est de minimiser temps global de l'exécution. Ceci peut être fait d'une manière :
  - ✓ **statique** au moment de la compilation ou au démarrage de programme (exemple : par estimation des temps d'exécution des tâches).
  - ✓ **dynamique** durant l'exécution du programme (exemple : utilisation d'un pool de tâches dans l'état prêt).
- Le parallélisme de tâches peut aussi être fourni au niveau langage de programmation qui spécifie le degré du parallélisme de taches disponible. Ceci a l'avantage que le programmeur est responsable seulement du degré de parallélisme qu'il a choisi, le reste est laissé au compilateur et au SE.

Distribution de données pour les tableaux

- Plusieurs algorithmes, et en particulier ceux de l'analyse numérique et de calcul scientifique, sont basés sur les vecteurs et les matrices.
- Les programmes correspondants utilisent des tableaux unidimensionnel, bidimensionnel ou de dimension supérieure comme structures de données.
- Une stratégie de parallélisation pour de tels programmes consiste en la décomposition du tableau de données en sous-tableaux et de les affecter aux différents processeurs : c'est la **distribution** (décomposition ou partitionnement) **de données**.
- La distribution de données peut être utilisée pour les machines à mémoire distribuée (DMM) et celle à mémoire partagée (SMM).
- Dans le cas des DMM, les données assignées à un processeur résident dans sa mémoire locale et ne peuvent être accédées que par ce dernier, minimisant ainsi les communications.
- Dans le cas des SMM, les processeurs accèdent à différentes parties de données ce qui empêche les conflits (concurrence et sections critiques)

Distribution de tableaux unidimensionnels

- La distribution de données pour les tableaux unidimensionnels se fait à l'aide de trois techniques : **la distribution par blocs, la distribution cyclique et distribution cyclique par blocs**.

- **La distribution par blocs** d'un tableau *v = (v1, . . . , vn)* de longueur *n* fait un découpage du tableau en *p* blocs contenant $\lceil n/p \rceil$ éléments consécutifs (*p* étant le nombre de processeurs).

- Le bloc *J* contient les éléments $(j-1).\lceil n/p \rceil +1, \ldots, j.\lceil n/p \rceil$ et il est affecté au processeur *Pj.*

- Lorsque *n* n'est pas un multiple de *p*, le dernier bloc contient moins de $\lceil n/p \rceil$ éléments.

- **Exemple :** pour *n=14* et *p=4* on a :

  P1 contient *v1, v2, v3, v4; P2* contient *v5, v6, v7, v8; P3* contient *v9, v10, v11, v12* et *p4* contient *v13,v14.*

Distribution de tableaux unidimensionnels (2)

- On peut aussi mettre $\lceil n/p \rceil$ éléments dans les *n mod p* premiers processeurs et $\lfloor n/p \rfloor$ éléments dans le reste des processeurs.

- **La distribution cyclique** d'un tableau unidimensionnel affecte les éléments du tableau aux processeurs par l'algorithme de tourniquet, c-à-d, l'élément *vi* est affecté au processeur $P_{(i-1) \bmod p + 1}$ $(i=1,\ldots,n)$.

- Le processeur *j* contient les éléments *j, j+p, …, j+p.(⌈n/p⌉-1)* pour *j≤n mod p. et j, j+p, …, j+p.(⌈n/p⌉-2) pour n mod p < j ≤ p.*

- **Exemple :** pour *n=14* et *p=4* on a :

  P1 contient *v1, v5, v9, v13; P2* contient *v2, v6, v10, v14; P3* contient *v3, v7, v11* et *p4* contient *v4,v8, v12.*

- **Distribution cyclique par blocs :** on peut combiner les deux types de distribution **cyclique et par blocs**. Dans ce cas, les éléments consécutifs sont structurés en blocs de taille b (b<< n/p). Les différents blocs sont ensuite affectés au processeurs d'une manière cyclique.

Distribution de tableaux bidimensionnels

- Pour les tableaux bidirectionnels, la combinaison des distributions cyclique et par blocs sont utilisées sur seulement une ou sur les deux dimension du tableau.

- Pour **la distribution sur une seule dimension**, les lignes ou les colonnes sont distribuées d'une manière cyclique, par blocs ou en combinant les deux modes.

- La **décomposition par blocs** des lignes/colonnes construit p blocs (de même taille) de linges/colonnes contigües. Les blocs sont ensuite affectés aux p processeurs (bloc i au processeur Pi).

- La **distribution cyclique** des lignes/colonnes affecte les lignes/colonnes au processeurs en utilisant l'algorithme de tourniquet.

- La **distribution cyclique par blocs** construit des blocs de b lignes/colonnes et les affecte aux processeurs d'une manière cyclique.

Distribution de tableaux bidimensionnels (2)

- La distribution des éléments d'un tableau de dimension *n1x n2* dans les deux dimensions se fait par une **distribution en échiquier** dans laquelle les processeurs sont organisés en une maille virtuelle de taille *p1x p2* .

- Les éléments *(k, i)* du tableau sont affectés aux processeurs *Pi, j , i = 1,..., p1, j = 1,.., p2.* La distribution en échiquier se fait également selon trois modes (par blocs, cyclique et cyclique par blocs).

- Dans le cas de la **distribution en échiquier par blocs :**
  - ✓ Le tableau est décomposé en *p1x p2* blocs d'éléments (le premier indice est décomposé en *p1* blocs et le deuxième indice est décomposé en *p2* blocs).
  - ✓ Le bloc *(i,j), 1≤i≤p1, 1≤j≤p2* est affecté au processeur dans la position *(i,j)* sur la maille virtuelle de processeurs.
  - ✓ Le bloc *(i,j)* contient les éléments *(k,l)*, tels que $k = (i-1)\cdot\lceil n1/p1 \rceil+1, \ldots, i\cdot\lceil n1/p1 \rceil$ et $l = (j-1)\cdot\lceil n2/p2 \rceil+1, \ldots, j\cdot\lceil n2/p2 \rceil$.

Distribution de tableaux bidimensionnels (3)

- Dans le cas de la **distribution en échiquier cyclique :**
  - ✓ Les éléments du tableau sont affectés au processeurs (dans la maille virtuelle de processeurs) par l'algorithme de tourniquet en affectant de manière **cyclique** les indices de lignes (resp. colonnes) du tableau aux lignes (resp. colonnes) de la maille de processeurs.
  - ✓ L'élément du tableau *(k, l)* est affecté au processeur dont la position dans la maille virtuelle de processeurs est $((k - 1) \bmod p1 + 1, (l - 1) \bmod p2 + 1)$
  - ✓ Autrement dit, on construit des blocs (sous-tableaux) de dimension *p1x p2* et on affecte l'élément *(i,j)* de chaque bloc au processeur dans la position *(i,j)* dans la maille de processeurs.

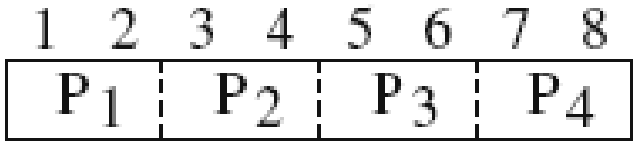
Distribution de tableaux bidimensionnels (4)

- Dans le cas de la **distribution en échiquier cyclique par blocs :**
  - ✓ On affecte des blocs de taille *b1xb2* d'une manière cyclique, tel que l'élément du tableau *(n,m)* appartient au bloc *(k,l)*, $k = \lceil m/b1 \rceil$ et $l = \lceil n/b2 \rceil$.
  - ✓ Le bloc *(k, l)* est affecté au processeur à la position $((k-1) \bmod p1+1, (l-1) \bmod p2+1)$.

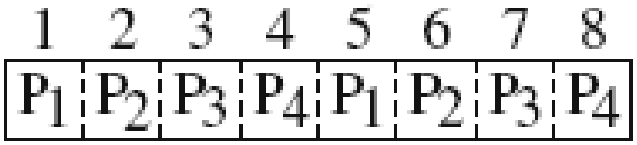
- **Remarque :** dans le cas des **tableaux de dimension supérieure** ($d>2$), on utilise une **distribution paramétrée** des éléments du tableau. C'est une généralisation de la distribution en échiquier 2D à une dimension quelconque (Pour plus de détails, voir le livre Parallel programming for multicore and cluster systems).

Exemple de distribution de tableaux 1D

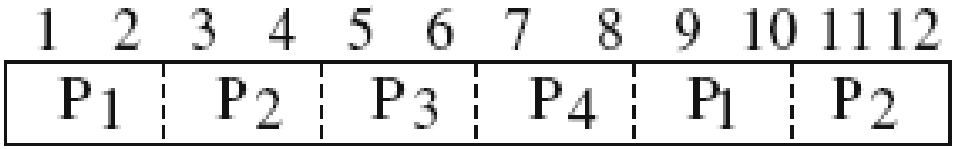
a) blockwise



cyclic

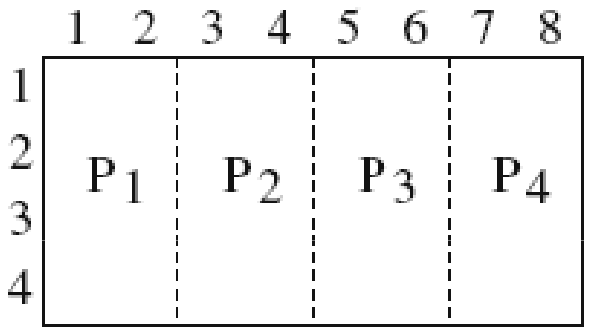


block–cyclic

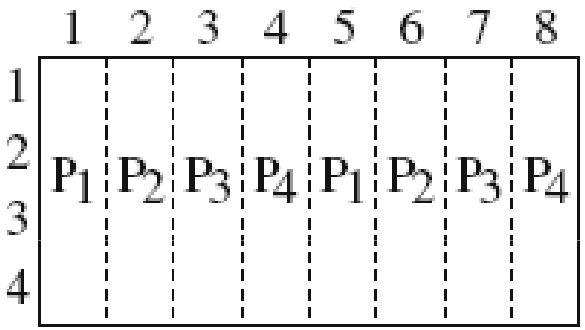


Exemple de distribution de tableaux 2D
(sur une dimension)

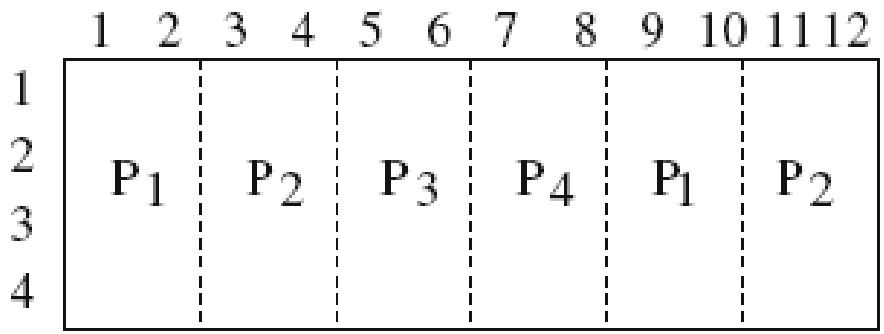
b) blockwise



cyclic

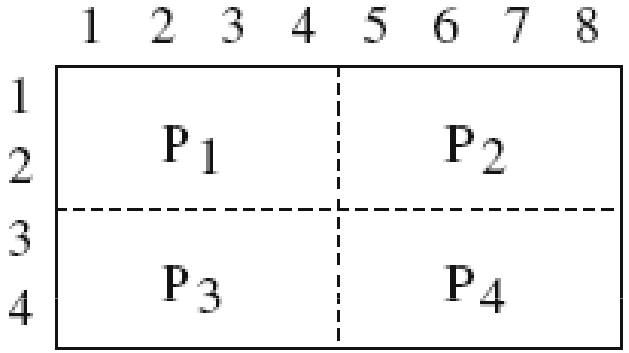


block–cyclic

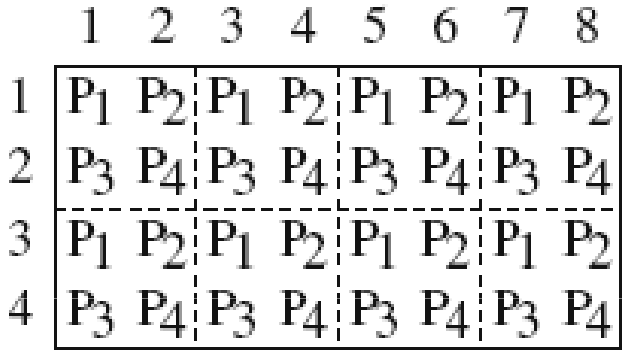


Exemple de distribution de tableaux 2D
(sur deux dimensions)

c) blockwise



cyclic



block–cyclic