Introduction au parallèlisme

Gérald MONARD

Bibliographie

– Parallel Programming in C with MPI and OpenMP
  Michael J. Quinn
  Ed. McGraw-Hill, New York - 2004.
– Tutorials from Lawrence Livermore National Laboratory
  `http ://www.llnl.gov/computing/tutorials/`
– Tutorials from Ohio Supercomputer Center
  `http ://www.osc.edu/hpc/training/`
– High Performance Computing, 2 nd edition
  Kevin Dowd and Charles Severance
  Ed. O'Reilly, Sebastopol, California, USA - 1998.

– Informatique parallèle et systèmes multiprocesseurs
  Jean-Louis Jacquemin
  Ed. Hermès, Paris - 1993.
– MPI - The Complete Reference : Volume 1, The MPI Core
  Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongara
  Ed. MIT Press, Massachusetts, USA - 1999.
– How to Build a Beowulf : A Guide to the Implementation and Application of PC Clusters
  Thomas L. Sterling, John Salmon, Donald J. Becker, Daniel F. Savarese
  Ed. MIT Press, Massachusetts, USA - 1999.

1. Introduction - Terminologie

1.1. Introduction

Traditionnellement, les logiciels sont écrits pour effectuer des calculs en *série* ou *séquentiels*. Ils sont exécutés sur un seul ordinateur ne possédant qu'un seul processeur. Les problèmes sont résolus par une série d'instructions qui sont exécutées les unes après les autres (séquentiellement) par le processeur. Seulement une seule instruction peut être exécutée à un moment donné dans le temps.

a). Qu’est-ce que le parallèlisme ?

Dans un sens général, le *parallèlisme* peut être défini comme une technique qui permet d’utiliser simultanément de multiples ressources de calculs afin de résoudre un problème informatique.

Ces ressources peuvent être :
– un seul ordinateur possédant plusieurs processeurs
– un certain nombre d’ordinateurs connectés entre par un réseau
– une combinaison des deux[a]

[a] Dans le reste du cours, nous nous limiterons à ce type de ressources et nous ne parlerons pas, par exemple, du parallèlisme à l’intérieur même des processeurs (co-processeurs, vectorisation, pipeline, *etc*).

En général, un problème informatique traité parallèlement possède des caractéristiques particulières telles que la possibilité :

– d’être découpé en plusieurs parties qui peuvent être résolues simultanément
– d’exécuter plusieurs programmes d’instructions à un même moment donné
– de résoudre plus rapidement le problème en utilisant de multiples ressources de calculs qu’en utilisant une seule ressource.

b). Pourquoi utiliser le parallèlisme ?

Il y a deux raisons principales qui incitent à utiliser le parallélisme :
– gagner du temps (*wall clock time* = temps mis pour effectuer une calcul)
– résoudre de plus gros problèmes
Les autres motivations possibles comprennent aussi :
– utiliser des ressources non locales (provenant d'un réseau plus vaste voire même d'Internet → Grid Computing)
– réduction des coûts
par ex. : utilisation de multiple ressources informatiques peu cher à la place de payer du temps de calculs sur un super-ordinateur → cluster Beowulf
– contrainte de mémoire : un seul ordinateur possède une taille de mémoire disponible finie, utiliser plusieurs ordinateurs peut permettre l'accès à une taille de mémoire globale plus importante

Le parallèlisme peut aussi être utilisé pour palier les limites/inconvénients des ordinateurs mono-processeurs :

– miniaturisation : de plus en plus de transistors dans un volume fini
– économie : il coûte de plus en plus cher de developper un processeur rapide et il peut être plus “rentable” d’utiliser un grand nombre de processeurs moins cher (et moins rapide) pour obtenir la même rapidité.

c). Le futur

Depuis une quinzaine d’années, les tendances en informatique indiquées par la conception de réseaux de plus en plus rapides, de systèmes distribués, d’architectures multi-processeurs, montre clairement que le parallélisme, sous toutes ses formes, est le présent mais aussi l’avenir du calcul informatique.

1.2. Terminologie

Tâche Une section finie d'instruction de calculs. Une tâche est typiquement un programme ou un jeu d'instructions qui est exécuté par un processeur

Tâche parallèle Une tâche qui peut être exécuté correctement par plusieurs processeurs (*i.e.* qui donne un résultat correct)

Exécution séquentielle Exécution d'un programme séquentiellement, *i.e.* une instruction à la fois. En premier lieu, cela correspond à ce qui se passe sur une machine mono-processeur. Cependant, virtuellement toutes les tâches parallèles ont des parties qui doivent être exécutés séquentiellement.

Exécution parallèle Exécution d'un programme par plus d'une tâche, avec chaque tâche capable d'exécuter la même ou différentes séries d'instructions au même moment dans le temps.

Mémoire partagée D'un point de vue matériel, cela correspond à une architecture d'ordinateurs où tous les processeurs ont un accès direct à une mémoire physique commune (généralement à travers le bus). D'un point de vue logiciel, cela décrit un modèle où toutes les tâches parallèles ont la même "image" de la mémoire et peuvent directement adresser et accèder les mêmes adresses logiques quel que soit l'endroit où se situe la mémoire physique.

Mémoire distribuée D'un point de vue matériel, cela correspond à un accès non commun à la mémoire physique (généralement à travers le réseau). D'un point de vue logiciel, les tâches ne peuvent "voir" que la mémoire locale de la machine et doivent utiliser des communications pour accéder la mémoire des autres machines et qui est utilisée par les autres tâches.

Communications Typiquement, les tâches parallèles ont besoin d'échanger des données. Il y a plusieurs façons de le faire, par exemple à travers une mémoire partagée ou un réseau. Mais dans un terme général on parle de communications quel que soit le moyen utilisé.

Synchronisation La coordination des taches parallèles en temps réel. Très souvent associé aux communications. Souvent implémenté par l'établissement de point de synchronisation dans une application où une tâche ne peut plus continuer tant que les autres tâches n'ont pas atteint le même point (ou un équivalent).

Granularité une mesure qualitative du rapport calcul sur communication. Le *gros grain* correspond à un grand nombre de calculs entre chaque communication. Le *grain fin* correspond à un petit nombre de calculs entre chaque communication.

Speed-up ou accélération. C'est, pour un code parallélisé, le rapport entre le temps d'exécution séquentiel (pour effectuer une tâche donnée) et le temps d'exécution parallèle (pour cette même tâche).

$$\text{Speed-up}(n) = \frac{\text{Temps séquentiel}}{\text{Temps parallèle (sur } n \text{ processeurs)}}$$

Parallel overhead ou surcoût parallèl. C’est le temps nécessaire pour coordonner les tâches parallèles (non utilisé pour effectuer du calcul). Celui-ci peut inclure :

– le temps de démarrage des tâches
– les synchronisations entre les différentes tâches parallèles
– les communications de données
– le surcoût logiciel dû aux compilateurs parallèles, aux bibliothèques, au système d’exploitation, etc.
– la terminaison des tâches.

Massivement parallèle cela se réfère à un système parallèle comprenant de nombreux processeurs, couramment plus de mille (1000).

Scalability ou propriété de croissance. Cela correspond à un système parallèle (matériel ou logiciel) qui possède la propriété suivante : son speed-up augmente lorsqu'on augmente le nombre de processeurs impliqués dans le parallèlisme. Les facteurs qui influent la "scalabilité" :

- le matériel (essentiellement la bande passante mémoire/processeur et les communications réseau)
- l'algorithme utilisé
- le surcoût parallèle associé
- les caractéristiques spécifiques de l'appplication et du codage associé

1.3. Problèmes associés au parallèlisme

Les problèmes liés au parallèlisme peuvent être regroupés en quatre catégories :

1. les difficultés dûs à l'aspect communication et qui font intervenir des problèmes fondamentaux de topologie, de routage et de blocage mutuel
2. les difficultés liées à la concurrence, comme l'organisation des mémoires, la cohérence de l'information et les problèmes de famine
3. les problèmes de coopération (décomposition du problème, équilibrage des charges, placement)
4. le non-déterminisme et les problèmes de terminaison

a). Les problèmes de communication :

Le temps consacré à la communication entre processeurs ou entre processeurs et mémoires est un des paramètres fondamentaux de l’efficacité des algorithmes parallèles.

Les temps de communications peuvent être décomposés en :

le temps de traitement de la communication c'est le temps nécessaire pour préparer l'information en vue de la transmission, *i.e.* assembler cette information en paquets, ajouter l'information de contrôle et l'adresse, sélectionner le lien, placer le paquet dans la mémoire tampon appropriée.

temps d'attente de transmission lorsque le paquet a été placé dans la mémoire tampon appropriée, l'information doit attendre que le lien soit disponible, ou bien que soit résolu un éventuel problème de contention, ou encore que la ressource requise soit disponible.

Dans certains systèmes, les échanges d'information se font sans tampon mais sont synchronisés selon la technique du rendez-vous. Les temps d'attente dans de tels systèmes ne sont donc pas des files d'attentes mais des attentes de synchronisation.

temps de transmission c'est le temps nécessaire à la transmission de tous les bits du paquet.

temps de propagation c'est le temps qui s'écoule entre l'émission du dernier bit du paquet et la réception de ce même bit.

On distingue deux grandes méthodes de communication qui influe principalement sur le temps de transmission :

les méthodes synchrones dans lesquelles les processeurs attendent en des points déterminés l'exécution de certains calculs ou l'arrivée de certaines données en provenance d'autres processeurs. Ces méthodes ont évidemment un grand impact sur les performances du système

les méthodes asynchrones où les processeurs n'ont plus aucune contraintes d'attente en des points déterminés mais où, en revanche, il est nécessaire de prendre de grandes précautions pour s'assurer de la validité du traitement. En effet, lorsqu'un processeur lira, par exemple, des données en provenance d'un autre processeur, il sera nécessaire de s'assurer qu'il ne s'agit pas de données obsolètes, c'est-à-dire antérieures à la dernière mise à jour, ou encore de données arbitraires dont la valeur n'aura pas encore été calculée par le processeur partenaire.

b). Topologie d'interconnexion des processeurs

Au plan matériel, une des grosses difficulté du parallèlisme est celle du choix de la topologie d'interconnexion, plus simplement appelée *topologie*. L'idéal serait de relier tous les processeurs deux à deux (connectivité totale) de façon à optimiser la communication entre tout processeur et tout autre processeur. Malheureusement, ceci n'est pas matériellement possible car le nombre de liaisons qu'un processeur peut avoir avec son environnement est limité. Il faut alors établir un compromis entre, d'une part, l'efficacité des communications, et, d'autre part, le coût de la connectique et la limitation du nombre de liaisons par processeurs.

Si plusieurs chemins sont possibles entre deux processeurs cherchant à communiquer, il est souhaitable d'optimiser la communication par un choix judicieux du chemin le plus efficace. Les algorithmes chargés de cette optimisation sont appelés *algorithmes de routage*.

Quelques exemples de topologie possible pour une architecture parallèle :

– en bus : simple, multiple, ou encore hiérarchisé. Peut permettre la mise en commun de la mémoire.
– réseau totalement connecté. Possible si le nombre de processeurs reste petit.
– pipeline ou réseau linéaire, il faut alors que la communication soit bidirectionnelle.
– anneau. Si les communications sont bidirectionnelles, il permet de maintenir la communication entre deux processeurs au cas où une connexion deviendrait défaillante.
– étoile. Le processeur central peut être source de conflits d'accès. Le nombre de liens nécessaires au processeur central croît avec la taille du réseau. C'est un système très mal adapté à la tolérance de panne.
– grille de processeurs (2D, 3D), tore, *etc*.
– hypercube
– en arbre. Permet de hiérarchiser les processeurs.

c). Routage

Dans un réseau d’interconnexion, un algorithme de routage permet aux paquets d’information d’être guidés à travers le réseau vers leur destination. Le principal objectif d’un tel algorithme est de sélectionner des chemins qui optimisent le temps que met chaque paquet pour se rendre du nœud source au nœud destination. Ce n’est pas forcément le plus court chemin (en cas de traffic important par exemple) et il peut différer d’un paquet à l’autre même si les terminaisons (processeurs) sont les mêmes.

d). Blocage mutuel

Cela peut arriver dans le cas où tous les processeurs se mettent en attente d'une information qui ne vient pas. Par exemple, si chaque nœud d'un réseau tente de transmettre de l'information au nœud opposé en même temps que tous les autres, il y aura blocage puisque chaque nœud, occupé à envoyer de l'information, ne pourra en même temps en recevoir.

e). Organisation de la mémoire

Deux types de problèmes :

– *Conflit d'accès* dans un système à mémoire partagée. La plupart du temps, ce problème doit être résolu au niveau logiciel.

– dans un système à mémoire distribuée, il peut y avoir une perte d'efficacité (**parallel overhead**) dû à l'augmentation du nombre de communications avec la croissance du nombre de processeurs. Le système consacre alors une part de plus en plus importante de son temps de traitement à des échanges de données entre processeurs plus ou moins distants (et donc pas en temps de calculs).

f). Cohérence de l’information

Dans le cas d’une mémoire partagée il est nécessaire de synchroniser les différents caches mémoires[a] des processeurs avec la mémoire principale.

La cohérence de la mémoire peut être gérée au niveau matériel (voir plus loin) soit au niveau logiciel (compilateur, programme utilisateur).

[a] Cache : dispositif matériel (usuellement une mémoire rapide) ou logiciel stockant dans une zone d’accès rapide des données qui se trouvent en grande quantité dans une zone beaucoup plus vaste mais d’accès plus lent.

g). Famine

La famine peut être matérielle. Elle est alors définie comme l'impossibilité pour un processeur P1 d'accéder à une ressource R ou de communiquer avec un autre processeur P2 du fait que R ou P2 sont eux-mêmes sollicités par d'autres processeurs et ne sont jamais à l'écoute de P1.

Mais la famine peut être aussi logicielle lorsqu'un processus p1 en sollicite un autre p2 mais ce dernier est lui-même sollicité par d'autres processus auxquels il donne toujours la priorité.

h). Equilibrage des charges - Décomposition

Cela consiste à équilibrer les tâches parallèles de manière à ce que leur temps d'exécution soit identique (ou presque). Cela ne peut s'effectuer que grâce à une bonne décomposition des tâches parallèles.

i). Non déterminisme

La mise en parallèle de plusieurs processus peut donner naissance à des problèmes d'un type particulier liés au non-déterminisme du comportement d'une application.

Exemple : le cas simple de 3 processus parallèles P1, P2 et P3, où P1 lit deux données D2 et D3 provenant respectivement de P2 et P3, et effectue seulement le traitement de la première donnée reçue. Le résultat dépend de la première donnée reçue par P1 et peut donc être totalement imprévisible.

j). Terminaison

La terminaison désigne l’action de mettre fin à (ou de conclure) un programme.

Terminer un programmme parallèle n’est pas une action triviale. Il est pourtant indispensable qu’il se termine proprement.

Par exemple, un programme parallèle ne se terminera pas proprement si l’une des tâches parallèles boucles indéfiniment, ou encore si tous les processus sauf un reçoivent l’ordre de “se terminer”.

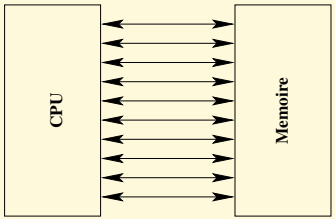
2. Architectures Parallèles

2.1. Machines Mono-processeurs

a). Architecture de von Neumann

Pendant plus de 40 ans, pratiquement tous les ordinateurs ont suivi un modèle commun de machine connu sous le nom de *modèle de von Neumann*, du nom du mathématicien hongrois John von Neumann.

Un ordinateur de von Neumann utilise le concept du programme stocké en mémoire. Le CPU exécute un programme stocké en mémoire et qui spécifie une séquence d'opérations de lecture et d'écriture dans la mémoire.



Architecture de von Neumann

Principales caractéristiques :

– La mémoire est utilisée à la fois pour stocker les instructions du programme et des données.
– Les instructions du programme sont des données codées qui disent à l'ordinateur ce qu'il faut faire.
– Les données sont simplement des informations destinées à être utilisé par le programme.
– Le CPU (*Central Processing Unit*) obtient les instructions ou les données de la mémoire, décode les instructions et les exécute *séquentiellement*.

b). Processeurs

Le processeur est le composant critique de calcul dans les ordinateurs.

De par leur architecture, les processeurs peuvent être divisés en trois familles :

CISC Complex Instruction Set Computer, qui utilisent un jeu nombreux d'instructions primitives puissantes, de taille variable et d'assez haut niveau.

Exemples :
– Motorola MC68000 (premiers Apple MacIntosh, Amiga, Atari ST)
– Début de la famille des x86

RISC Reduced Instruction Set Computer, qui utilisent un jeu réduit d'instructions primitives de bas niveau et de taille unique.

Exemples :
– MIPS R8000 (utilisé dans les Silicon Graphics)
– DEC Alpha 21164 (Cray T3E)
– SUN UltraSPARC

Post-RISC qui augmentent les performances des processeurs RISC, entre autres, en exécutant des instructions hors contexte (*out-of-order execution*) et des instructions spéculatives (*speculative execution*). Les exécutions hors contexte consistent à exécuter des instructions qui sont tardives dans le code avant celle qui viennent plus tôt quand cela permet d'utiliser plus efficacement le processeur. Les exécutions spéculatives consistent à commencer le calcul après un branchage avant que le test de celui-ci soit effectué (typiquement relatif à une instruction `if`).

Exemples :

– Intel Pentium IV
– AMD Athlon/Opteron
– IBM Power 4+
– MIPS R16000

c). Mémoires

La performance de la mémoire est aussi importante que la performance du processeur, car ce dernier obtient ses instructions et des données du premier.

Il existe deux quantités primordiales qui décrivent les mémoires :

– le temps d'accès mémoire, qui donne le temps nécessaire pour lire ou écrire en mémoire
– le temps de cycle mémoire, qui décrit la fréquence à laquelle on peut réaccéder à une référence en mémoire.

Le principal problème est que la mémoire travaille en général moins rapidement que le processeur.

Solution : l'utilisation de plusieurs niveaux de caches qui sont des mémoires plus rapides, mais donc plus coûteuses et donc présentes en moins grandes quantités.

Exemple : temps d’accès mémoire sur un DEC Alpha 21164 500 Mhz

Registres	2 ns
L1 (sur le proc.)	4 ns
L2 (sur le proc.)	5 ns
L3 (en dehors du proc.)	30 ns
Mémoire principale	220 ns

L’un des problèmes importants survenant dans l’utilisation des caches est la réactualisation des données caches/mémoires. Si le processeur change la donnée dans le cache, cette donnée doit aussi changer dans les mémoires de niveau supérieur. Cela se fait grâce au *mapping* qui met en relation les locations mémoires et le cache.

d). Organisation des caches mémoires

Le cache peut être organisé selon plusieurs manières :

direct mapped l'algorithme le plus simple où le cache contient directement une partie de la mémoire.

Par exemple, si le cache est de taille 4 Ko, l'adresse mémoire 0 est directement relié à l'adresse 0 du cache, ainsi que 4 Ko, 8 Ko, 12 Ko, *etc*.

Evidemment, le direct mapping pose problème lorsque l'on veut accèder à la fois à des parties de la mémoire qui sont distantes (en adressage).

fully associative où toute adresse mémoire peut directement être recopié (“mappé”) dans le cache. A chaque fois qu’une demande de donnée faite par le processeur ne peut être accompli par le cache, le cache libère une place (habituellement LIFO, *Last In, First Out*) et recopie la donnée de la mémoire dans le cache.

Les caches complètement associatifs sont supérieur en utilisation par rapport au direct mapping. Cependant ils sont très coûteux en terme de prix, taille et rapidité (*i.e.*, il faut d’abord regarder si la donnée est dans le cache, puis déterminer quelle partie à libérer puis faire la recopie).

set associative cela correspond à l’association de plusieurs direct mapped cache, généralement deux (two-way set associative) ou quatre (four-way set associative). Dans ce cas, on ne libère qu’une partie (way) du cache à la fois (LIFO sur le way). C’est cette gestion du cache qui est un compromis en terme de rapidité, coût et efficacité du cache, et qui est habituellement utilisé dans les processeurs à hautes performances.

2.2. Machines Multi-processeurs

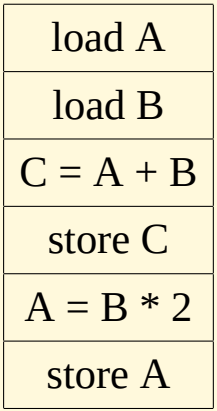
a). Taxonomie de Flynn

Il existe différentes façons de classifier les ordinateurs parallèles. L'une des classifications les plus utilisées (depuis 1966) est appelée Taxonomie de Flynn. Elle fait la distinction entre les architectures multi-processeurs selon deux dimensions indépendantes : les instructions et les données. Chaque dimension ne peut avoir qu'un seul état : simple ou multiple. La matrice suivante définit les 4 classements possibles selon Flynn :

SISD Instruction simple, Donnée simple	SIMD Instruction simple, Donnée multiple
MISD Instruction multiple, Donnée simple	MIMD Instruction multiple, Donnée multiple

SISD Single Instruction, Single Data

– ordinateur séquentiel
– une seule instruction : un seul flot d'instructions est exécuté par un seul processeur à un moment donné
– une seule donnée : un seul flot de donné est utilisé à un moment donné
– exécution déterministe
– la plus vieille forme d'ordinateur
– exemples : PC, Stations de travail (mono-processeur), *etc*.



SIMD Single Instruction, Multiple Data

– un type d'ordinateur parallèle
– une seule instruction : tous les unités de calcul exécute la même instruction à n'importe quelle cycle d'horloge
– multiple donnée : chaque unité de calcul peut opérer sur un élément de donnée différent

prev instruct	prev instruct	…	prev instruct
load A(1)	load A(2)		load A(n)
load B(1)	load B(2)		load B(n)
C(1)=A(1)*B(1)	C(2)=A(2)*B(2)	…	C(n)=A(n)*B(n)
store C(1)	store C(2)		store C(n)
next instruct	next instruct	…	next instruct

– Ce type de machine a typiquement un répartiteur d'instructions et un réseau interne très rapide.
– idéal pour des problèmes très spécifiques et caractérisés par un haut degré de régularité, comme par exemple le traitement d'images.
– Deux grandes variétés d'ordinateurs : les ordinateurs vectoriels (Cray C90, Fujitsu VP, NEC SX-5, *etc*) et les matrices de processeurs (Connection Machine CM-2, Maspar MP-1, MP-2, *etc*).

MISD Multiple Instruction, Single Data

– très peu d'exemples d'ordinateurs de cette classe
– exemples possibles : filtres à fréquences multiples agissant sur le même flot de données, ou de multiples algorithmes de cryptographie essayant de "craquer" le même message codé.

MIMD Multiple Instruction, Multiple Data

– Actuellement, le type d'ordinateurs parallèles le plus couramment rencontré
– instruction multiple : chaque processeur peut exécuter un flot d'instructions différents
– donnée multiple : chaque processeur peut travailler sur un flot de données différent

prev instruct	prev instruct	…	prev instruct
load A(1)	call funcD		do i=1,N
load B(1)	x=y*z		alpha=w**3
C(1)=A(1)*B(1)	sum=x*2	…	zeta=C(i)
store C(1)	call sub1(i,j)		end do
next instruct	next instruct	…	next instruct

– l’exécution peut être synchrone ou asynchrone, déterministe ou non-déterministe
– Exemples : la plupart des “super-ordinateurs” actuels (SPx, Origin 3x00, HP Superdome, *etc*), les grilles d’ordinateurs reliés entre eux par un réseau, les ordinateurs SMP (*Symmetric MultiProcessor*), *etc*.

b). Machines à mémoire partagée

L’architecture des machines à mémoire partagée varie grandement, mais elles possèdent en commun la possibilité pour tous les processeurs d’accéder à toute la mémoire de façon globale (adressage global).

Les multiples processeurs peuvent opérer indépendamment, mais partage les mêmes ressources mémoires.

Le changement de la mémoire à un endroit par un processeur est visible par tous les autres processeurs.

Les machines à mémoire partagée peuvent être divisées en deux grandes classes basées sur le temps d'accès mémoire :

Uniform Memory Access (UMA)

- représentée essentiellement aujourd'hui par les machines SMP
- les processeurs sont identiques
- accès égal à la mémoire
- temps d'accès mémoire identique pour chaque processeur
- Parfois appelé CC-UMA pour *Cache Coherent UMA* dans le cas où si un processeur rafraîchi une location mémoire, tous les autres processeurs le savent. La cohérence au niveau du cache est effectuée au niveau matériel.

Non-Uniform Memory Access (NUMA)

- Souvent réalisée en reliant physiquement deux ou plus de machines SMP
- Un SMP peut accéder directement à la mémoire d'un autre SMP
- Tous les processeurs n'ont pas un égal temps d'accès mémoire
- l'accès à la mémoire à travers le lien physique est plus lent
- si la cohérence de cache est maintenue, on parle de CC-NUMA.

Avantages des machines à mémoire partagée :

– L’adressage globale de la mémoire permet une programmation simplifié de la gestion mémoire
– le partage des données entre les tâches est à la fois rapide et uniforme dû à la proximité mémoire/CPU.

Inconvénients :

– Le principal inconvénient est le manque de “scalabilité” entre la mémoire et les CPUs. L’addition de processeurs peut augmenter le traffic de données, et pour les systèmes à cache cohérent, cela peut augmenter le temps de gestion cache/mémoire.
– Coût : il est de plus en plus difficile d’inventer et de construire des machines à mémoire partagée avec un nombre toujours croissant de processeurs.

c). Machines à mémoire distribuée

Comme les machines à mémoire partagée, les architectures de machines à mémoire distribuée varie grandement, mais partage une caractéristique commune : elle nécessite un réseau de communication pour connecter les mémoires inter-processeurs.

Les processeurs ont chacun leur propre mémoire locale. Les adresses mémoires sur un processeur ne correspondent pas avec celle d'un autre processeur. Il n'y a donc pas de concept de mémoire globale pour tous les processeurs.

Comme chaque processeur possède sa propre mémoire, il agit indépendamment des autres. Les changements à la mémoire sur un processeur n'affecte pas la mémoire des autres processeurs. Il n'y a donc pas de concept de cohérence de cache.

Quand un processeur a besoin d'accéder aux données d'un autre processeur, il est habituellement du ressort de l'utilisateur de définir explicitement comment et quand les données seront communiquées. De même, la synchronisation entre les tâches est de la responsabilité du programmeur.

Avantages des machines à mémoire distribuée :

– La mémoire croît avec le nombre de processeurs. Augmenter le nombre de processeurs permet d'augmenter proportionnellement la mémoire adressable.
– Chaque processeur peut accéder très rapidement à sa propre mémoire sans interférence et sans le surcoût imposé par la maintenance de la cohérence de cache.
– Coût global : on peut utiliser des processeurs et des réseaux de base, rendant ainsi l'ordinateur parallèle peu coûteux ("Beowulf")

Inconvénients :

– Le programmeur est responsable de nombreux détails concernant la communication de données entre les processeurs.
– NUMA

d). Machines Hybrides

Les ordinateurs parallèles les plus gros et les plus performants actuellement emploient à la fois les architectures tirés de la mémoire distribuée et de la mémoire partagée.

Le composant mémoire partagée est généralement une machine SMP cache cohérente. Les processeurs sur un SMP donné peuvent accéder globalement à la mémoire de cette machine

Le composant mémoire distribuée est un réseau de multiples machines SMP. Les SMPs ne connaissent que leur mémoire et donc, le réseau de communication est nécessaire pour transférerles données d'un SMP vers un autre.

La tendance semble montrer que ce type d'architecture va continuer à prévaloir dans les années à venir.

e). Exemples de machines parallèles

Architecture	CC-UMA	CC-NUMA	Distribuée
Exemples	AMD Opterons Sun Fire IBM Power4/5	SGI Altix 3000 SGI Origin 3900 HP Superdome	Cray XD1 IBM BlueGene/L IBM SP4/5
Scalabilité	quelques proc.	centaines de proc.	milliers de proc.

Top 500 des machines parallèles dans le monde : `http://www.top500.org/`

3. Parallèlisme logiciel

3.1. Modèle de programmation

Il existe plusieurs modèles de programmation parallèles qui sont employés régulièrement :
– Mémoire partagée (Shared Memory)
– Threads
– Passage de messages (Message Passing)
– Données parallèles (Data Parallel)
Les modèles de programmation parallèles existent comme une abstraction au-delà de l'architecture matériel et de l'architecture mémoire.

a). Mémoire partagée

Dans ce modèle, les tâches partagent un même espace d'adressage dans lequel elles lisent et écrivent de manière asynchrone. Divers mécanismes (sémaphores, *etc*) permettent de contrôler l'accès à la mémoire partagée. L'avantage de ce modèle est qu'il y manque la notion d'appartenance des données, et donc il n'est pas toujours nécessaire de spécifier explicitement la communication de données entre tâches (tout le monde possède la donnée). Par contre, un inconvénient sérieux est qu'il est difficile de comprendre et de maintenir la localité des données.

Pas d'implémentation standard.

b). Threads

Dans ce modèle, un processus simple peut avoir de multiples chemins d'exécution concurrents.

Par exemple, une tâche peut s'exécuter séquentiellement puis créer un certain nombre de tâches (filles), les *threads* [a], qui vont être exécutées de manière concurrentielle (parallèlement) par le système d'exploitation.

Le modèle par les threads est généralement associé aux architectures à mémoire partagée.

Implémentations : cela comprend une bibliothèque de routines qui sont appelées à l'intérieur du code parallèle et un jeu de directives de compilation insérées dans le code source séquentiel ou parallèle.

Dans ce modèle, le programmeur est responsable de la détermination de tout le parallèlisme.

[a] Traduction littérale : fil (de nylon).

Deux standards : POSIX Threads et OpenMP.

POSIX Threads

- Basé sur une bibliothèque de routines.
- Nécessite un codage parallèle explicite
- Standard IEEE (1995)
- Langage C seulement
- Parallèlisme très explicite (bas niveau) et oblige le programmeur à être très attentif au détail de programmation.

Open MP

- Basé sur des directives de compilation.
- Peut utiliser du code séquentiel
- Supporté par un consortium de vendeurs de compilateurs et de fabriquants de matériels.
- Portable, multiplateformes (inclus UNIX et Windows NT)
- Existe pour C, C++ et Fortran
- Peut être très simple à manipuler.
- `http ://www.openmp.org`

c). Passage de message

Le modèle par passage de messages possède les caractéristiques suivantes :

– un ensemble de tâches utilise leur mémoire local propre pour effectuer les calculs. Ces tâches résident sur la même machine physique ou sont réparties sur un nombre arbitraire de machines.

– Les tâches échangent des données à travers des communications en envoyant ou en recevant des messages

– Le transfert de données requiert des opérations coopératives qui doivent être effectuées par chaque processus.
Par exemple, une opération d'envoi doit être associée à une opération de réception.

Implémentations La plupart du temps, cela recouvre une bibliothèque de routines que l'on inclut dans le code source. Le programmeur est responsable de la détermination de l'ensemble du parallèlisme.

Ancêtre : PVM, Parallel Virtual Machine. Cette implémentation comprend la notion de machine virtuelle dans laquelle les processus communiquent. Cette machine virtuelle est répartie sur un réseau d'ordinateurs qui peut être hétérogène.

Actuellement, la forme la plus utilisée du passage de messages est MPI (Message Passing Interface) qui a été créé par un consortium (MPI Forum) regroupant utilisateurs, constructeurs et fabriquant logiciel (compilateurs).

`http ://www.mcs.anl.gov/Projects/mpi/standard.html`

`http ://www.mpi-forum.org`

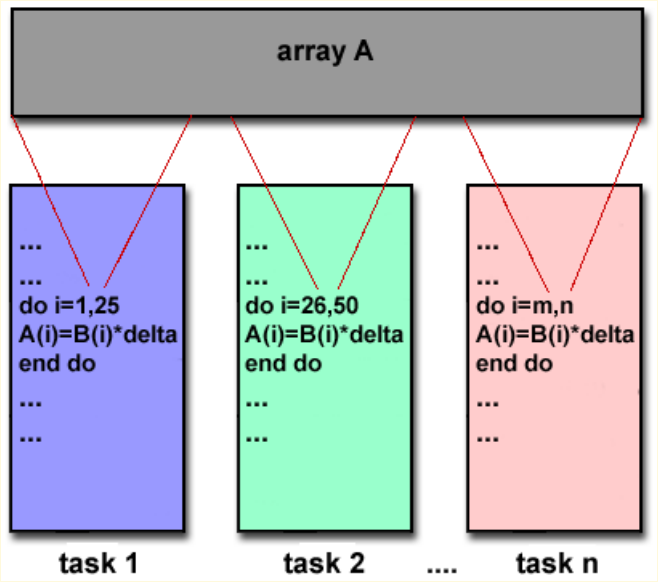
C'est le standard "de facto" industriel. Deux versions : MPI-1 (1992-1994) et MPI-2 (1995-1997).

Dans un système à mémoire partagée, les implémentations de MPI n'utilise pas le réseau pour communiquer les données mais utilisent des tampons partagées (copie de mémoire).

d). Donnée parallèle

Le modèle à donnée parallèle possède les caractéristiques suivantes :

– Le travail parallèle principal regroupe essentiellement les opérations sur un jeu de donnée. Celui-ci est typiquement organisé dans une structure commune, comme une matrice ou un cube.
– Un jeu de tâche travaille collectivement sur la même structure de donnée, même si chaque tâche travaille sur une différente partition de cette structure de donnée.
– Les tâches effectue la même opération sur leur partition de travaille (exemple : ajouter 4 aux éléments d’un tableau).



Dans une architecture à mémoire partagée, toutes les tâches ont accès à la structure de données à travers la mémoire globale. Dans une architecture à mémoire distribuée, la structure de donnée est divisée en “morceaux” qui sont réparties dans la mémoire locale de chaque tâche.

Implémentations Programmer avec le modèle de donnée parallèle est habituellement accompli en écrivant un programme avec des *constructeurs* de données parallèles. Ces constructeurs peuvent être des appels à des routines d'une bibliothèque parallèle ou des directives de compilations reconnues par le compilateur.

Fortran 90 standard ISO/ANSI provenant de Fortran 77.

– Contient complètement Fortran 77 (compatible)
– Nouveau format d'écriture du code
– Additions faites aux structures de programme et aux commandes
– Pointeurs, allocation de mémoire dynamique
– Calcul sur les matrices (considérées comme des objets)
– Récursivité + nouvelles fonctions intrinsèques

Il existe des implémentations Fortran 90 pour architecture parallèle qui tiennent compte du parallèlisme de données.

High Performance Fortran (HPF) Extensions à Fortran 90 pour supporter la programmation en parallèlisme de données.

– Contient tout Fortran 90
– Directives pour indiquer au compilateur comment distribuer les données
– Ajout d'assertions qui peuvent améliorer l'optimisation du code compilé
– Constructeur de données parallèles (inclus dans Fortran 95)

3.2. Introduction à OpenMP

a). OpenMP

`http ://www.openmp.org`

OpenMP est :

– une API (Application Program Interface) qui permet un parallèlisme explicite en mémoire partagée utilisant le modèle des Threads.
– possède trois composantes :
  – Directives de compilation
  – Bibliothèques de routines pour l'exécution
  – Variables d'environnement
– portable. L'API existe pour les languages C, C++ et Fortran, et pour la plupart des environnements UNIX (et Windows NT)
– un standard (provient d'un consortium)

OpenMP n’est pas :

– utilisable dans un parallèlisme à mémoire distribuée
– toujours implémenté de manière identique par tous les vendeurs (malgré le fait que ce soit un standard !)
– garanti d’utiliser le plus efficacement possible la mémoire partagée.

b). But d’OpenMP

– Offrir un standard de programmation (bientôt ANSI ?)
– Etablir un jeu de directives simples et limitées pour le parallèlisme à mémoire partagée
– Etre facile à utiliser
– Permettre à la fois le grain fin et le gros grain
– Portabilité

c). OpenMP : Modèle de programmation

– Basé sur les threads
– Parallèlisme explicite, mais non automatique. Le programmeur a le contrôle total sur la parallèlisation
– Modèle ”Fork-Join” :
  – Tout programme OpenMP commence comme un seul processus : le **maître**. Ce maître est exécuté séquentiellement jusqu’à ce une **région parallèle** est rencontrée
  – **FORK :** le maître (un thread) crée alors une **équipe** de threads parallèles
  – les déclarations dans la région parallèle du programme sont exécutées en parallèle par l’équipe de threads
  – **JOIN :** lorsque le travail des threads est terminé dans la région parallèle, ceux-ci se synchronisent, se terminent, et laissent seul le maître continuer l’exécution du programme

– Quasiment tout le parallèlisme est spécifié grâce à des directives de compilation inclues dans le code source
– Possibilité d'imbriquer des régions parallèles à l'intérieur de régions parallèles (défini dans l'API mais peu ou pas disponible dans les compilos)
– Le nombre de threads peut varier de manière dynamique

d). OpenMP : Code général

Fortran :

```
      PROGRAM HELLO
      INTEGER VAR1, VAR2, VAR3

      Serial Code
            .
            .
            .
      Beginning of paralle section. Fork a team of threads.
      Specify variable scoping

!$OMP PARALLEL PRIVATE(VAR1,VAR2) SHARED(VAR3)

      Parallel section executed by all threads.
            .
            .
            .
      All threads join master thread and disband

!$OMP END PARALLEL

      Resume Serial Code
            .
            .
            .
      END
```

e). OpenMP : Code général

C :

```
#include <omp.h>

main()
{
   int var1, var2, var3;

      Serial Code
            .
            .
            .
      Beginning of paralle section. Fork a team of threads.
      Specify variable scoping

#pragma omp parallel private(var1, var2) shared(var3)
   {
      Parallel section executed by all threads.
            .
            .
            .
      All threads join master thread and disband
   }

      Resume Serial Code
            .
            .
            .
}
```

f). OpenMP : Directives

Format Fortran :

sentinelle	**directive**	**[clause]**

`!$OMP`
`C$OMP`
`*$OMP`
	un nom valide	option

Example :

```
C$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
```

Format C :

`#pragma omp`	**directive**	**[clause]**	**retour chariot**
Obligatoire	un nom valide	option	obligatoire

Example :

```
#pragma omp parallel default(shared) private(beta,pi)
```

g). OpenMP : régions parallèles

Une région parallèle est un bloc de codes qui doit être exécuté par une équipe de threads.

Fortran :

```
!$OMP PARALLEL [clause ...]
!$OMP&  IF (scalar_logical_expression)
!$OMP&  PRIVATE (list)
!$OMP&  SHARED (list)
!$OMP&  DEFAULT (PRIVATE | SHARED | NONE)
!$OMP&  REDUCTION (operator: list)

      block

!$OMP END PARALLEL
```

C :

```
#pragma omp parallel [clause ...]     \
      if (scalar_logical_expression) \
        private (list)                 \
        shared (list)                  \
        default (shared | none)        \
        reduction (operator: list)

structured_block
```

h). OpenMP : régions parallèles

Lorsqu'un processus/thread rencontre une directive PARALLEL, il crée une équipe de threads et devient le maître de cette équipe. Il possède alors le numéro 0 dans cette équipe.

A partir du début de la région parallèle, le code est dupliqué et toutes les threads vont l'exécuter.

Il y a une barrière implicite à la fin de la région parallèle. Seul le maître continue l'exécution après.

Le nombre de threads dans la région est déterminé par :

1. la fonction `omp_set_num_treads()`
2. la variable d'environnement `OMP_NUM_THREADS`
3. l'implémentation par défaut

Les threads sont numérotés de 0 à N-1.

⟶ voir programmes `hello`

i). Partage de travail

Un partage de travail peut s'effectuer entre les différents membres d'une équipe.

Type de partage :

DO/For : les itérations sont réparties sur les threads (parallèlisme de données)

SECTIONS : le travail est séparé en partie indépendantes. Chaque section est exécuté par une thread. (parallèlisme fonctionnel)

SINGLE exécute séquentiellement une section de code

j). OpenMP : DO

```
!$OMP DO [clause ...]
         SCHEDULE (type [,chunk])
         PRIVATE (list)
         SHARED (list)
         REDUCTION (operator | intrinsic : list)

    do_loop

!$OMP END DO   [ NOWAIT ]
```

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                private (list)
                shared (list)
                reduction (operator: list)
                nowait

  for_loop
```

k). OpenMP : DO

SCHEDULE description de la façon dont la boucle doit être divisée
- STATIC
- DYNAMIC

NOWAIT pas de synchronisation des threads à la fin de la boucle

Restriction :
- Pas de `DO WHILE` ou de boucle sans contrôle d'arrêt.
- La variable d'itération doit être entière et la même pour toutes les threads
- Le programme doit être correct quelque soit le thread qui exécute un morceau de la boucle.
- Pas de branchage en dehors de la boucle

⟶ `vecteuradd`

l). SECTIONS

Cela correspond à une partie non-itérative de travail. La directive inclut différentes sections qui sont réparties selon les différents threads.

```
!$OMP SECTIONS [clause ...]
                PRIVATE (list)
                REDUCTION (operator | intrinsic : list)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

```
#pragma omp sections [clause ...]  newline
                    private (list)
                    firstprivate (list)
                    lastprivate (list)
                    reduction (operator: list)
                    nowait
{

#pragma omp section    newline

   structured_block

#pragma omp section    newline

   structured_block

}
```

Il y a une barrière implicite à la fin de la directive `SECTIONS`, sauf en présence de `NOWAIT`.

$\longrightarrow$ `vecteuradd2`

m). SINGLE

Cette directive spécifie que le code ne doit être exécuté que par une seule thread.

Cela peut servir pour les opérations I/O

```
!$OMP SINGLE [clause ...]
             PRIVATE (list)
             FIRSTPRIVATE (list)

   block

!$OMP END SINGLE [ NOWAIT ]
```

```
#pragma omp single [clause ...]  newline
                   private (list)
                   firstprivate (list)
                   nowait

     structured_block
```

n). PARALLEL DO

Combine à la fois la déclaration d'une région PARALLEL et d'une boucle répartie DO

```
!$OMP PARALLEL DO [clause ...]
                  IF (scalar_logical_expression)
                  DEFAULT (PRIVATE | SHARED | NONE)
                  SCHEDULE (type [,chunk])
                  SHARED (list)
                  PRIVATE (list)
                  REDUCTION (operator | intrinsic : list)

   do_loop

!$OMP END PARALLEL DO
```

```
#pragma omp parallel for [clause ...]  newline
                         if (scalar_logical_expression)
                         default (shared | none)
                         schedule (type [,chunk])
                         shared (list)
                         private (list)
                         reduction (operator: list)

   for_loop
```

$\longrightarrow$ `vecteuradd3`

De même il existe `PARALLEL SECTIONS`

o). BARRIER

Point de synchronisation. Aucune tâche ne continue tant que toutes les autres ne sont pas arrivées au même point.

```
!$OMP BARRIER
```

```
#pragma omp barrier newline
```

p). REDUCTION

⟶ voir `reduction`

q). OpenMP :Routines

OMP_SET_NUM_THREADS

```
SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
```

```
void omp_set_num_threads(int num_threads)
```

OMP_GET_NUM_THREADS

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

```
int omp_get_max_threads(void)
```

OMP_GET_THREAD_NUM

```
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

```
int omp_get_thread_num(void)
```

3.3. Introduction à MPI

a). MPI

`http ://www.mpi-forum.org`

MPI est une spécification d'interface
MPI = Message Passing Interface

C'est une spécification pour les développeurs et les utilisateurs. Ce n'est pas une bibliothèque en soi (mais plutôt des instructions sur comment la bibliothèque devrait être).

Conçu pour être un standard pour le parallèlisme en mémoire distribuée et à passage de messages

Existe pour C et Fortran

Tout le parallèlisme est explicite

Avantages de MPI :

– Standard. Est utilisable sur quasiment toutes les platesformes existantes
– Portable
– Souvent performant, car les vendeurs peuvent exploités les spécificités du matériel dans leur implémentation
– Fonctionnel (plus de 115 routines)
– Acessible. Beaucoup d'implémentations, propriétaires ou libres.
– Peut être utilisé aussi sur des machines à mémoire partagée

b). MPI : Base

Un fichier d'entête est nécessaire dans chaque programme/routine qui fait appel à des routines MPI

```
include 'mpif.h'
#include "mpi.h"
```

Puis on peut faire appel à des routines MPI :

```
CALL MPI_XXXXX(parameter,..., ierr)
CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)
rc = MPI_Xxxxx(parameter, ... )
rc = MPI_Bsend(&buf,count,type,dest,tag,comm)
```

c). MPI :Structure Générale

```
MPI include file
.
.
Initialisation de l'environnement MPI
.
.
Travail, passage de messages
.
.
Terminaison de l'environnement MPI
```

Chaque processus, sous MPI, possède un identifier (ID) et un rang (de 0 à N-1). Il appartient à une communauté (“Communicator”)

d). MPI : Environnement

MPI_Init Initialise l'environnement MPI. C'est la première fonction à appeler. On ne le fait qu'une seule fois.

MPI_Comm_Size Détermine le nombre de processus dans une communauté. Utilisé avec `MPI_COMM_WORLD` qui représente la communauté entière des processus, cela permet d'obtenir le nombre total de processus utilisé dans l'application.

MPI_Comm_rank Détermine le rang dans une communauté du processus appelant.

MPI_Abort Termine tous les processus associés à la communauté.

MPI_Finalize Termine l'environnement MPI. C'est la dernière fonction à appeler dans tout programme MPI.

```c
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[]; {
int  numtasks, rank, rc;

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf ("Number of tasks=%d My rank=%d\n", numtasks,rank);

/*******  do some work *******/

MPI_Finalize();
}
```

```
      program simple
      include 'mpif.h'

      integer numtasks, rank, ierr, rc

      call MPI_INIT(ierr)
      if (ierr .ne. MPI_SUCCESS) then
         print *,'Error starting MPI program. Terminating.'
         call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
      end if

      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
      print *, 'Number of tasks=',numtasks,' My rank=',rank

C ****** do some work ******

      call MPI_FINALIZE(ierr)

      end
```

e). MPI : Arguments

Les routines de communication point-à-point on généralement le format suivant :

Envoi bloquant

```
MPI_Send(buffer,count,type,dest,tag,comm)
```

Envoi non bloquant

```
MPI_Isend(buffer,count,type,dest,tag,comm,request)
```

Reception bloquante

```
MPI_Recv(buffer,count,type,source,tag,comm,status)
```

Reception non bloquante

```
MPI_Irecv(buffer,count,type,source,tag,comm,request)
```

– Buffer : adresse de tampon qui référence les données à envoyer/recevoir
– Count : nombre d'éléments à envoyer/recevoir.
– Type : type d'éléments. MPI prédéfini différents types (mais l'utilisateur peut en définir d'autres) :

`MPI_CHAR`	signed char
`MPI_SHORT`	signed short int
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_UNSIGNED_INT`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double
...	...

– Destination : le rang du destinataire
– Source : le rang de l'envoyeur (`MPI_ANY_SOURCE` correspond à recevoir un message de n'importe quel processus)
– Tag : un entier arbitraire non négatif défini par le programmeur et qui identifie uniquement le message. Les opérations de réception et d'envoi doivent avoir le même "tag"
(`MPI_ANY_TAG` correspond à n'importe quel identifiant).
– Communicator : précise le contexte de communauté dans laquelle se fait le message.
– Status : structure (C) ou vecteur (Fortran) qui précise la source d'un message et le tag correspondant. Grâce au status, on peut aussi obtenir le nombre de bytes reçus.
– Request : permet dans le cas d'une opération non-bloquante si l'envoi/réception est terminé.

```c
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
                tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
                tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
                tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
                tag, MPI_COMM_WORLD);
  }

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);


MPI_Finalize();
```

```
      program ping
      include 'mpif.h'

      integer numtasks, rank, dest, source, count, tag, ierr
      integer stat(MPI_STATUS_SIZE)
      character inmsg, outmsg
      tag = 1

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      if (rank .eq. 0) then
         dest = 1
         source = 1
         outmsg = 'x'
         call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
     &           MPI_COMM_WORLD, ierr)
         call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
     &           MPI_COMM_WORLD, stat, ierr)

      else if (rank .eq. 1) then
         dest = 0
         source = 0
         call MPI_RECV(inmsg, 1, MPI_CHARACTER, source, tag,
     &        MPI_COMM_WORLD, stat, err)
         call MPI_SEND(outmsg, 1, MPI_CHARACTER, dest, tag,
     &        MPI_COMM_WORLD, err)
      endif

      call MPI_GET_COUNT(stat, MPI_CHARACTER, count, ierr)
      print *, 'Task ',rank,': Received', count, 'char(s) from task',
     &           stat(MPI_SOURCE), 'with tag',stat(MPI_TAG)

      call MPI_FINALIZE(ierr)

      end
```

```
      program ringtopo
      include 'mpif.h'

      integer numtasks, rank, next, prev, buf(2), tag1, tag2, ierr
      integer stats(MPI_STATUS_SIZE,4), reqs(4)
      tag1 = 1
      tag2 = 2

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      prev = rank - 1
      next = rank + 1
      if (rank .eq. 0) then
         prev = numtasks - 1
      endif
      if (rank .eq. numtasks - 1) then
         next = 0
      endif

      call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1,
     &     MPI_COMM_WORLD, reqs(1), ierr)
      call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2,
     &     MPI_COMM_WORLD, reqs(2), ierr)

      call MPI_ISEND(rank, 1, MPI_INTEGER, prev, tag2,
     &     MPI_COMM_WORLD, reqs(3), ierr)
      call MPI_ISEND(rank, 1, MPI_INTEGER, next, tag1,
     &     MPI_COMM_WORLD, reqs(4), ierr)

      call MPI_WAITALL(4, reqs, stats, ierr);

      call MPI_FINALIZE(ierr)

      end
```

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

f). Communications Collectives

Elles concernent **tous** les processus à l'intérieur d'une communauté.

Il est de la responsabilité du programmeur de s'assurer que tous les processus dans cette communauté participent à la communication.

Types possibles :
– Synchronisation
– Mouvement de données (broadcast, gather, all to all)
– Calcul collectif (reduction)
Les opérations collectives sont bloquantes

Pas besoin de "tag"

Seulement des types de données prédéfinies par MPI, pas de types définis par l'utilisateur.

g). Communications Collectives

MPI_Barrier point de synchronisation

MPI_Bcast envoi d'un message à toute la communauté

MPI_Scatter distribution de messages distincts à chaque processus de la communauté (ex. : envoi d'un tableau, chaque proc. en reçoit une partie)

MPI_Gather inverse de Scatter. Réception de données distinctes provenant des processus de la communauté.

MPI_Reduce applique une opération de réduction sur tous les processus et place le résultat dans un seul.

```
      program scatter
      include 'mpif.h'

      integer SIZE
      parameter(SIZE=4)
      integer numtasks, rank, sendcount, recvcount, source, ierr
      real*4 sendbuf(SIZE,SIZE), recvbuf(SIZE)

C     Fortran stores this array in column major order, so the
C     scatter will actually scatter columns, not rows.
      data sendbuf /1.0, 2.0, 3.0, 4.0,
     &              5.0, 6.0, 7.0, 8.0,
     &              9.0, 10.0, 11.0, 12.0,
     &              13.0, 14.0, 15.0, 16.0 /

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      if (numtasks .eq. SIZE) then
         source = 1
         sendcount = SIZE
         recvcount = SIZE
         call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf,
     &  recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr)
         print *, 'rank= ',rank,' Results: ',recvbuf
      else
         print *, 'Must specify',SIZE,' processors.  Terminating.'
      endif

      call MPI_FINALIZE(ierr)

      end
```

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors.  Terminating.\n",SIZE);

MPI_Finalize();
}
```

3.4. Limites du parallèlisme

a). Loi d'Amdahl

Soit P la fraction parallèlisable d'un programme et S sa fraction non-parallèlisable (séquentielle). On a alors :

$$\text{speed-up}(n) = \frac{P+S}{\frac{P}{n}+S} = \frac{1}{P/n+S}$$

L'efficacité du parallèlisme est donc limité :

	speed-up		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.99	99.02

b). Loi de Gustavson

On suppose que la taille du problème croît linéairement avec le nombre de processeurs. La partie concernant les données est parallélisable, le reste ne l'est pas.

Temps séquentiel = s + a * n

Temps parallèle = s + a * n / n = s + a

$$\text{speed-up}(n) = \frac{s + a * n}{s + a}$$

qui tend vers l'infini lorsque n tend vers l'infini.

Conclusion des deux lois Le parallèlisme est limité par la taille de données (partie parallélisable), pour avoir une grande accélération il faut agir sur un volume important de données (voire croissant).