Optimisation de logiciels de modélisation sur centre de calcul

Gérald Monard

Pôle de Chimie Théorique
http://www.monard.info/

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 (modèle de von Neumann).

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[1]

[1] 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.

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.
- ☞ les dernières générations de puces Intel utilisent des multi-cores

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

Communications

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

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

les méthodes synchrones

les méthodes asynchrones

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.

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

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

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).

Machines Multi-processeurs: Taxonomie de Flynn

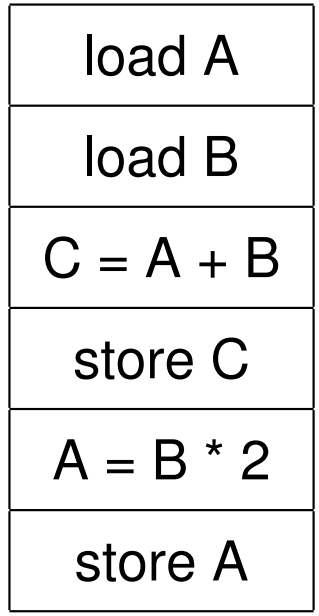
Il existe différentes façons de classifier les ordinateurs parallèles. L'une des classifications les plus utilisées (depuis 1966) est appelée Taxonomie de Flynn. Elle fait la distinction entre les architectures multi-processeurs selon deux dimensions indépendantes : les instructions et les données. Chaque 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 GPU (NVIDIA Tesla, 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, les grilles d'ordinateurs reliés entre eux par un réseau, les ordinateurs SMP (Symmetric MultiProcessor), etc.

Les différents types de machines parallèles

à mémoire partagée

à mémoire distribuée

hybride

Machines à mémoire partagée

- Les processeurs accèdent à toute la mémoire de façon globale (adressage global).
- Le changement de la mémoire à un endroit par un processeur est visible par tous les autres processeurs.
- Avantages: gestion simplifiée de la mémoire, partage des données
- Inconvénients: manque de “scalabilité”

Machines à mémoire distribuée

- Chaque processeur a sa 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.
- 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.

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.
- Ce sont des réseaux de multiples machines SMP.

Parallélisme logiciel: 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.

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, 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.

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`

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).

Introduction à OpenMP

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.

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

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

OpenMP : Code général (Fortran)

```
1       PROGRAM HELLO
2       INTEGER VAR1, VAR2, VAR3
3
4       Serial Code
5             .
6             .
7       Beginning of paralle section. Fork a team of threads.
8       Specify variable scoping
9
10 !$OMP PARALLEL PRIVATE(VAR1,VAR2) SHARED(VAR3)
11
12      Parallel section executed by all threads.
13            .
14            .
15      All threads join master thread and disband
16
17 !$OMP END PARALLEL
18
19      Resume Serial Code
20            .
21            .
22      END
```

OpenMP : Code général (C)

```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
                .
                .
}
```

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)
```

OpenMP : régions parallèles

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

Fortran :

```
1  !$OMP PARALLEL [clause ...]
2  !$OMP&  IF (scalar_logical_expression)
3  !$OMP&  PRIVATE (list)
4  !$OMP&  SHARED (list)
5  !$OMP&  DEFAULT (PRIVATE | SHARED | NONE)
6  !$OMP&  REDUCTION (operator: list)
7
8        block
9
10 !$OMP END PARALLEL
```

C :

```
1 #pragma omp parallel [clause ...]     \
2       if (scalar_logical_expression) \
3         private (list)                \
4         shared (list)                 \
5         default (shared | none)       \
6         reduction (operator: list)
7
8 structured_block
```

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_threads()`
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`

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

OpenMP : DO

```
1  !$OMP DO [clause ...]
2           SCHEDULE (type [,chunk])
3           PRIVATE (list)
4           SHARED (list)
5           REDUCTION (operator | intrinsic : list)
6
7      do_loop
8
9  !$OMP END DO   [ NOWAIT ]
```

```
1  #pragma omp for [clause ...]  newline
2                  schedule (type [,chunk])
3                  private (list)
4                  shared (list)
5                  reduction (operator: list)
6                  nowait
7
8    for_loop
```

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`

SECTIONS

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

```
1  !$OMP SECTIONS [clause ...]
2                 PRIVATE (list)
3                 REDUCTION (operator | intrinsic : list)
4
5  !$OMP  SECTION
6
7     block
8
9  !$OMP  SECTION
10
11     block
12
13 !$OMP END SECTIONS  [ NOWAIT ]
```

```
1  #pragma omp sections [clause ...]  newline
2                       private (list)
3                       firstprivate (list)
4                       lastprivate (list)
5                       reduction (operator: list)
6                       nowait
7  {
8
9  #pragma omp section    newline
10
11    structured_block
12
13 #pragma omp section    newline
14
15    structured_block
16
17 }
```

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

⟶ `vecteuradd2`

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
```

PARALLEL DO

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

```
1  !$OMP PARALLEL DO [clause ...]
2                    IF (scalar_logical_expression)
3                   DEFAULT (PRIVATE | SHARED | NONE)
4                   SCHEDULE (type [,chunk])
5                   SHARED (list)
6                   PRIVATE (list)
7                   REDUCTION (operator | intrinsic : list)
8
9     do_loop
10
11 !$OMP END PARALLEL DO
```

```
1 #pragma omp parallel for [clause ...]  newline
2                          if (scalar_logical_expression)
3                          default (shared | none)
4                          schedule (type [,chunk])
5                          shared (list)
6                          private (list)
7                          reduction (operator: list)
8
9    for_loop
```

⟶ `vecteuradd3`

De même il existe `PARALLEL SECTIONS`

BARRIER

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

```
!$OMP BARRIER
```

```
1 #pragma omp barrier  newline
```

REDUCTION

⟶ voir `reduction`

OpenMP :Routines

OMP_SET_NUM_THREADS

```
1 SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
```

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

OMP_GET_NUM_THREADS

```
1 INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

```
1 int omp_get_max_threads(void)
```

OMP_GET_THREAD_NUM

```
1 INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

```
1 int omp_get_thread_num(void)
```

Introduction à MPI

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

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)
```

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”)

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();
}
```

```
 1       program simple
 2       include 'mpif.h'
 3
 4       integer numtasks, rank, ierr, rc
 5
 6       call MPI_INIT(ierr)
 7       if (ierr .ne. MPI_SUCCESS) then
 8          print *,'Error starting MPI program. Terminating.'
 9          call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
10       end if
11
12       call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
13       call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
14       print *, 'Number of tasks=',numtasks,' My rank=',rank
15
16    C ****** do some work ******
17
18       call MPI_FINALIZE(ierr)
19
20       end
```

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_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_UNSIGNED_INT`	unsigned int
`MPI_FLOAT`	float
`MPI_DOUBLE`	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.

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main(argc,argv)
5  int argc;
6  char *argv[];  {
7  int numtasks, rank, dest, source, rc, count, tag=1;
8  char inmsg, outmsg='x';
9  MPI_Status Stat;
10
11 MPI_Init(&argc,&argv);
12 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
13 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
1  if (rank == 0) {
2    dest = 1;
3    source = 1;
4    rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
5                  tag, MPI_COMM_WORLD);
6    rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
7                  tag, MPI_COMM_WORLD, &Stat);
8    }
9
10 else if (rank == 1) {
11   dest = 0;
12   source = 0;
13   rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
14                 tag, MPI_COMM_WORLD, &Stat);
15   rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
16                 tag, MPI_COMM_WORLD);
17   }
18
19 rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
20 printf("Task %d: Received %d char(s) from task %d with tag %d \n",
21        rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
22
23
24 MPI_Finalize();
25 }
```

```
1       program ping
2       include 'mpif.h'
3
4       integer numtasks, rank, dest, source, count, tag, ierr
5       integer stat(MPI_STATUS_SIZE)
6       character inmsg, outmsg
7       tag = 1
8
9       call MPI_INIT(ierr)
10      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
11      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
```

```
 1       program ringtopo
 2       include 'mpif.h'
 3
 4       integer numtasks, rank, next, prev, buf(2), tag1, tag2, ierr
 5       integer stats(MPI_STATUS_SIZE,4), reqs(4)
 6       tag1 = 1
 7       tag2 = 2
 8
 9       call MPI_INIT(ierr)
10       call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
11       call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
12
13       prev = rank - 1
14       next = rank + 1
15       if (rank .eq. 0) then
16          prev = numtasks - 1
17       endif
18       if (rank .eq. numtasks - 1) then
19          next = 0
20       endif
```

```
 1        call MPI_IRECV(buf(1), 1, MPI_INTEGER, prev, tag1,
 2       &     MPI_COMM_WORLD, reqs(1), ierr)
 3        call MPI_IRECV(buf(2), 1, MPI_INTEGER, next, tag2,
 4       &     MPI_COMM_WORLD, reqs(2), ierr)
 5
 6        call MPI_ISEND(rank, 1, MPI_INTEGER, prev, tag2,
 7       &     MPI_COMM_WORLD, reqs(3), ierr)
 8        call MPI_ISEND(rank, 1, MPI_INTEGER, next, tag1,
 9       &     MPI_COMM_WORLD, reqs(4), ierr)
10
11        call MPI_WAITALL(4, reqs, stats, ierr);
12
13        call MPI_FINALIZE(ierr)
14
15        end
```

```
 1 #include "mpi.h"
 2 #include <stdio.h>
 3
 4 int main(argc,argv)
 5 int argc;
 6 char *argv[];  {
 7 int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
 8 MPI_Request reqs[4];
 9 MPI_Status stats[4];
10
11 MPI_Init(&argc,&argv);
12 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
13 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14
15 prev = rank-1;
16 next = rank+1;
17 if (rank == 0)  prev = numtasks - 1;
18 if (rank == (numtasks - 1))  next = 0;
```

```
1  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
2  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
3
4  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
5  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
6
7  MPI_Waitall(4, reqs, stats);
8
9  MPI_Finalize();
10 }
```

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.

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.

```
1         program scatter
2         include 'mpif.h'
3
4         integer SIZE
5         parameter(SIZE=4)
6         integer numtasks, rank, sendcount, recvcount, source, ierr
7         real*4 sendbuf(SIZE,SIZE), recvbuf(SIZE)
8
9   C     Fortran stores this array in column major order, so the
10  C     scatter will actually scatter columns, not rows.
11        data sendbuf /1.0, 2.0, 3.0, 4.0,
12       &           5.0, 6.0, 7.0, 8.0,
13       &           9.0, 10.0, 11.0, 12.0,
14       &           13.0, 14.0, 15.0, 16.0 /
15
16        call MPI_INIT(ierr)
17        call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
18        call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
```

```
1         if (numtasks .eq. SIZE) then
2            source = 1
3            sendcount = SIZE
4            recvcount = SIZE
5            call MPI_SCATTER(sendbuf, sendcount, MPI_REAL, recvbuf,
6        &    recvcount, MPI_REAL, source, MPI_COMM_WORLD, ierr)
7            print *, 'rank= ',rank,' Results: ',recvbuf
8         else
9            print *, 'Must specify',SIZE,' processors.  Terminating.'
10        endif
11
12        call MPI_FINALIZE(ierr)
13
14        end
```

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #define SIZE 4
4
5  int main(argc,argv)
6  int argc;
7  char *argv[];  {
8  int numtasks, rank, sendcount, recvcount, source;
9  float sendbuf[SIZE][SIZE] = {
10   {1.0, 2.0, 3.0, 4.0},
11   {5.0, 6.0, 7.0, 8.0},
12   {9.0, 10.0, 11.0, 12.0},
13   {13.0, 14.0, 15.0, 16.0}  };
14 float recvbuf[SIZE];
15
16 MPI_Init(&argc,&argv);
17 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
18 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

```
1  if (numtasks == SIZE) {
2    source = 1;
3    sendcount = SIZE;
4    recvcount = SIZE;
5    MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
6                MPI_FLOAT,source,MPI_COMM_WORLD);
7
8    printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
9           recvbuf[1],recvbuf[2],recvbuf[3]);
10   }
11 else
12   printf("Must specify %d processors. Terminating.\n",SIZE);
13
14 MPI_Finalize();
15 }
```