# Programmation parallèle et/ou distribuée

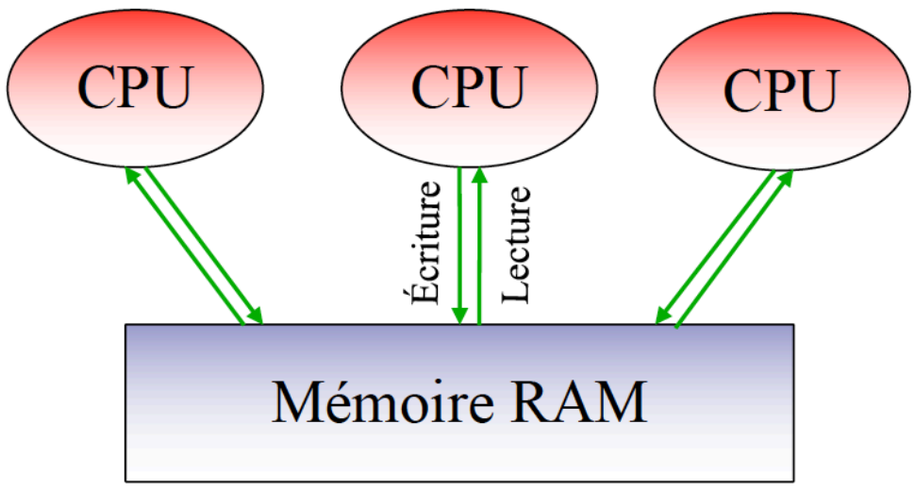
## Autres moyens

# Calcul parallèle : autre définition

« Faire coopérer plusieurs processeurs pour réaliser un calcul »

- Avantages:
  - Rapidité : pour N processeurs, temps de calcul divisé par N, en théorie…
  - Taille mémoire : pour N processeurs, on dispose de N fois plus de mémoire (en général)
- Difficultés:
  - Il faut gérer le partage des tâches.
  - Il faut gérer l'échange d’information (tâches non-indépendantes)

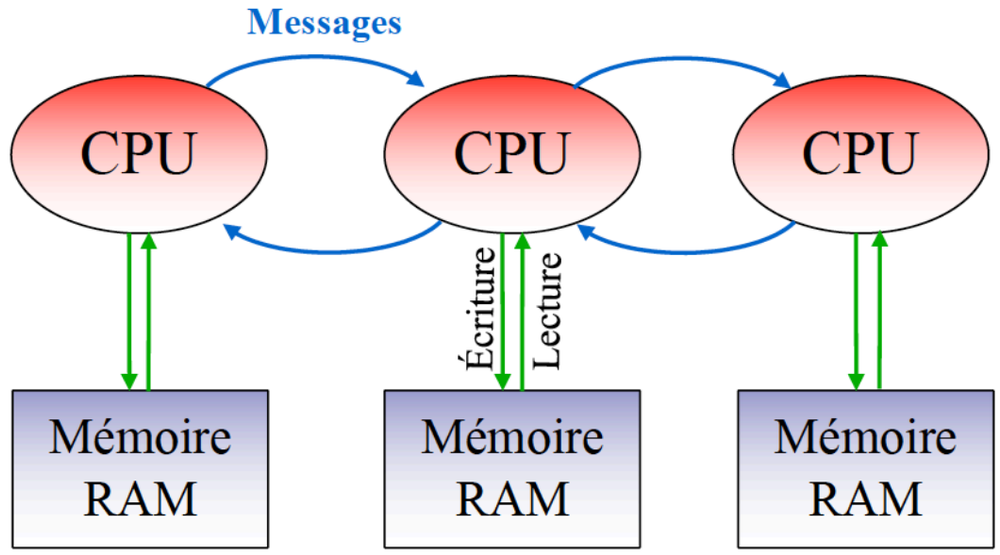
# Mémoire partagée / distribuée




## Mémoire partagée (SMP)

Tous les processeurs ont accès à l'ensemble de la mémoire.

→Attention aux **conflits**.
→Très peu de surcoût de parallélisation.



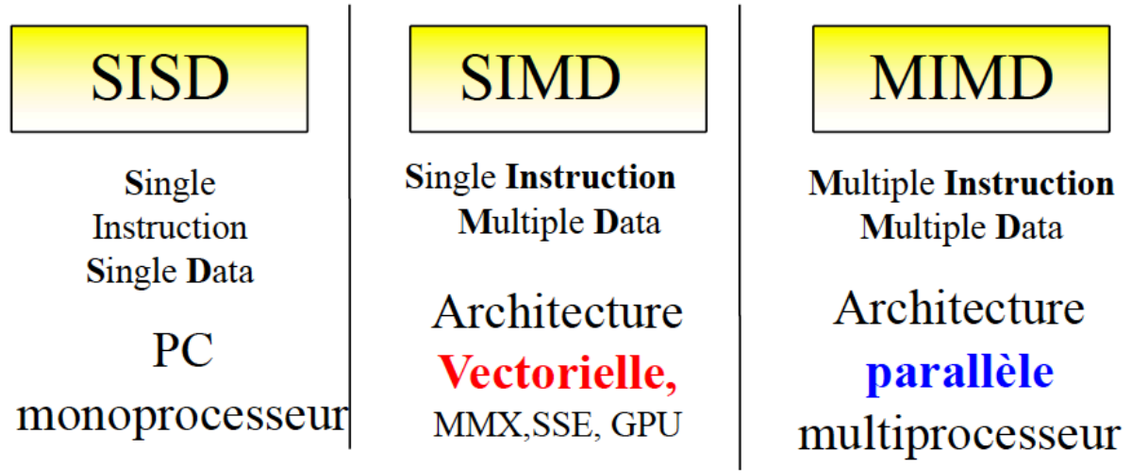

## Mémoire distribuée

Chaque processeur possède sa propre mémoire. Il n'a pas accès à celle des autres.

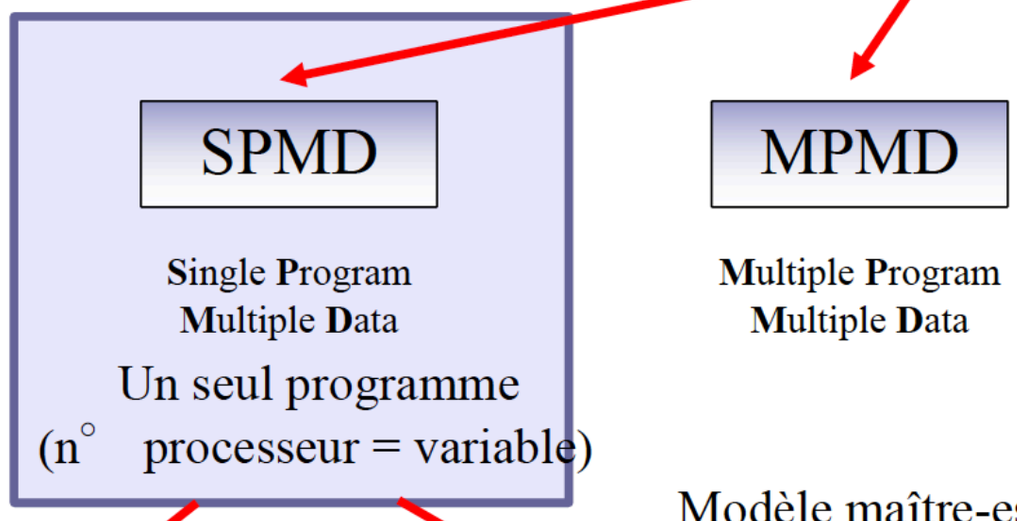
→Il faut gérer l'échange de messages (surcoût)

# Modèles de parallélisme

**Architecture matérielle:**

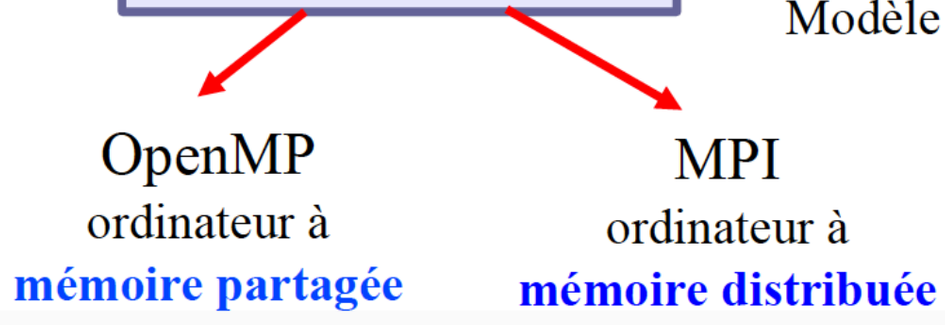



**Modèle de programmation:**




Modèle maître-esclave.

**Outils de parallélisation:**



# MPI : Message Passing Interface

Une bibliothèque de communication pour le parallélisme sur architectures à mémoire distribuée (Fortran, C, C++)

  - Valable aussi sur architecture à mémoire partagée

- Premiers standards MPI en 1994

- Un programme MPI est constitué de plusieurs taches (au sens processus), chaque tache a un espace mémoire propre non accessible directement aux autres

- Communication inter-taches (coordination) via l'échange explicit de messages (via une interface réseau)

- Les ressources (données en mémoire) sont locales (ou privées) à une tache

# MPI : exemple

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

#include <mpi.h>

int main(int argc, char* argv[])
{

  int nbTask;
  int myRank;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &nbTask);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("I am task %d out of %d\n",myRank,nbTask);

  MPI_Finalize();

  return 0;

}
```

# MPI : compilation et exécution

- Compiler
  - en C: mpicc -o helloworld_mpi helloworld_mpi.c
  - Pour savoir ce qui se cache derrière, tapper mpicc -showme : mpicc appelle un compilateur C qui peut être gcc, icc, pgcc, ...
- Exécuter:
  - ./helloworld_mpi
  - mpirun -np 2 ./helloworld_mpi
  - On peut lancer le programme avec plus de processus que de processeurs, toutefois pas recommandé

# MPI : exemple de communications

```
MPI_Send( message, 13, MPI_CHAR, i,  type, MPI_COMM_WORLD  );
MPI_Recv( message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
```

Type of data, should be same for send and receive
*MPI_Datatype type*

Number of elements (items, not bytes)
Recv number should be greater than or equal to amount sent
*int count*

Address where the data start
*void* data*

# MPI : exemple de communications

```
MPI_Send(  message, 13, MPI_CHAR, i,  type, MPI_COMM_WORLD );

MPI_Recv(  message, 20, MPI_CHAR, 0, type, MPI_COMM_WORLD, &status);
```

Identify process you're communicating with by rank number
*int dest/src*

Arbitrary tag number, must match up (receiver can specify MPI_ANY_TAG to indicate that any tag is acceptable)
*int tag*

Communicator specified for send and receive must match, no wildcards
*MPI_Comm comm*

Returns information on received message
*MPI_Status* status*

# MPI : communications collectives

`MPI_Bcast()` – Broadcast (one to all)

`MPI_Reduce()` – Reduction (all to one)

`MPI_Allreduce()` – Reduction (all to all)

`MPI_Scatter()` – Distribute data (one to all)

`MPI_Gather()` – Collect data (all to one)

`MPI_Alltoall()` – Distribute data (all to all)

`MPI_Allgather()` – Collect data (all to all)

# MPI : exemple de synchronisation

```
int main(int argc, char ** argv)
{
    MPI_Init(&argc, &argv);

    int wrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &wrank);

    if (wrank==0)
    {
        FILE* f = fopen("outin", "w");
        long int seconds = time(NULL);
        fprintf(f, "%ld", seconds);
        fclose(f);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    long int witness = 0;
    FILE* f = fopen("outin", "r");
    fscanf(f, "%d", &witness);
    printf("Rang %d, witness %ld.\n", wrank, witness);
    fclose(f);

    MPI_Finalize();
    return 0;
}
```

```
$ mpirun -n 4 source
Rang 1, witness 1450259857.
Rang 0, witness 1450259857.
Rang 2, witness 1450259857.
Rang 3, witness 1450259857.
```

# OpenMP

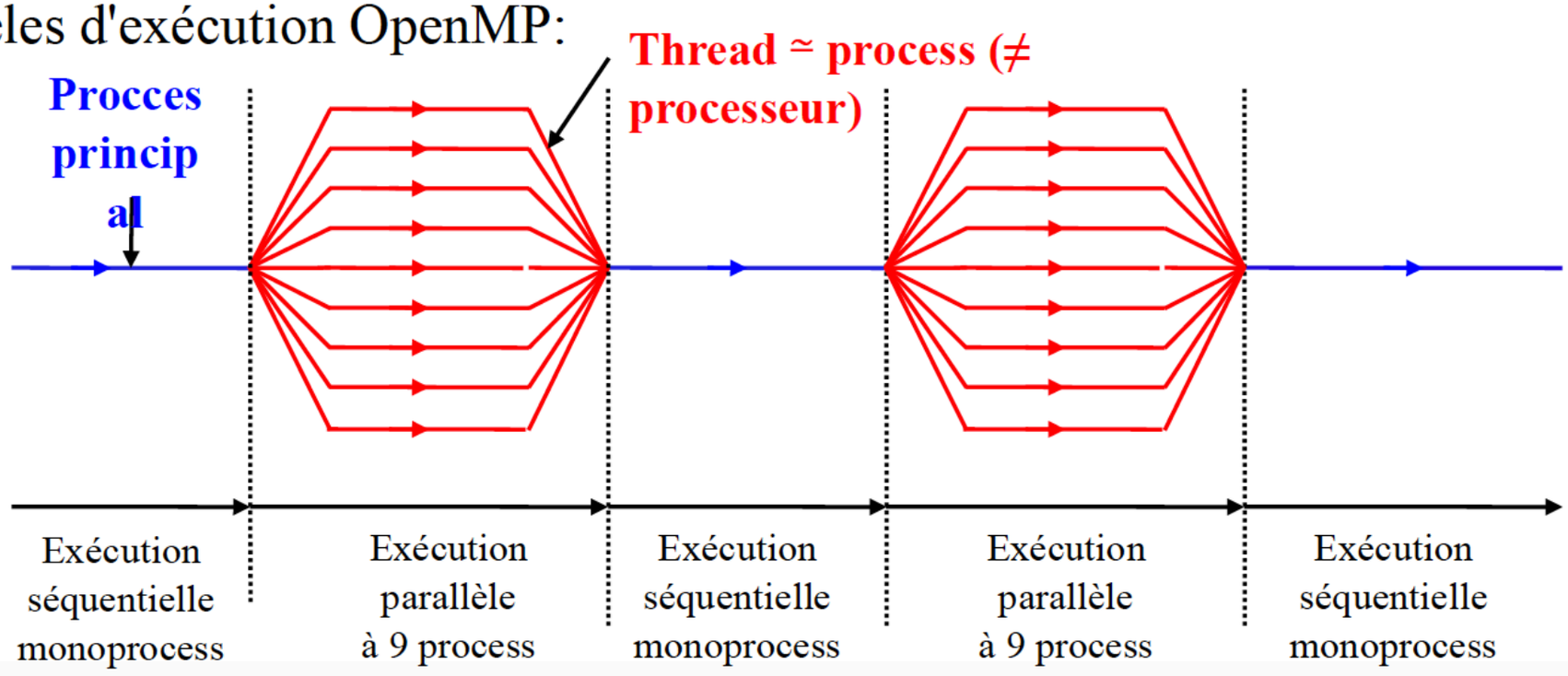
# OpenMP

OpenMP est un ensemble de directives de compilation pour paralléliser un code sur une architecture SMP (interfaces Fortran, C et C++)

**Le compilateur interprète** les directives OpenMP (si il en est capable!)

Les standards d'OpenMP datent de 1997, ceux d'OpenMP-2 de 2000, OpenMP-3 2008. Les développeurs de compilateurs les implémentent.

Modèles d'exécution OpenMP:



# OpenMP : produit scalaire (1/2)

```c
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  for (size_t i = 0; i < SIZE; i++)
    sum = sum + a[i]*b[i];

  printf("sum = %g\n", sum);
  return 0;
}
```

# OpenMP : produit scalaire (2/2)

```c
#include <stdio.h>
#define SIZE 256

int main() {
  double sum, a[SIZE], b[SIZE];

  // Initialization
  sum = 0.;
  for (size_t i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }

  // Computation
  #pragma omp parallel for reduction(+:sum)
  for (size_t i = 0; i < SIZE; i++) {
    sum = sum + a[i]*b[i];
  }

  printf("sum = %g\n", sum);
  return 0;
}
```

# OpenMP: compilation et exécution

- Compiler
  - Support dans la plupart des compilateurs classiques
  - En C : gcc -fopenmp prog.c -o prog

- Exécuter:
  - Positionnement éventuel des variables d’environnement ( OMP_NUM_THREADS, OMP_DYNAMIC, etc)
  - ./prog

# OpenMP : quelques directives

- Construction de régions parallèles
  - parallel : crée une région parallèle sur le modèle fork-join
- Partage du travail
  - for : partage des itérations d'une boucle parallèle
  - sections : définit des blocs à exécuter en parallèle
  - single : déclare un bloc à exécuter par un seul thread
- Synchronisation
  - master : déclare un bloc à exécuter par le thread maître
  - critical : bloc à n'exécuter qu'un thread à la fois
  - atomic : instruction dont l'écriture mémoire est atomique  barrier : attente que tous les threads arrivent à ce point

# OpenMP : exemple de synchronisation

La directive BARRIER synchronise les threads: tous s'arrêtent au niveau de la directive jusqu'à ce que le dernier soit arrivé. Puis ils continuent tous. Syntaxe:

**!$OMP BARRIER**

RDV

# Outils de parallelisation

- Langages ou extensions de langages
  - CUDA, OpenCL, etc.
- Bibliothèques:
  - Message Passing Interface (MPI), Pthread, IPC, etc.
- Directives de compilation:
  - OpenMP, directives d’accélération pour GPU, etc.
- Compilateurs (efficacité très faible)
- Outils
- MatLab, ….