Using cholesky easily. Part 2

Equipe IDCS

January 3 and 4, 2025

Institut Polytechnique de Paris

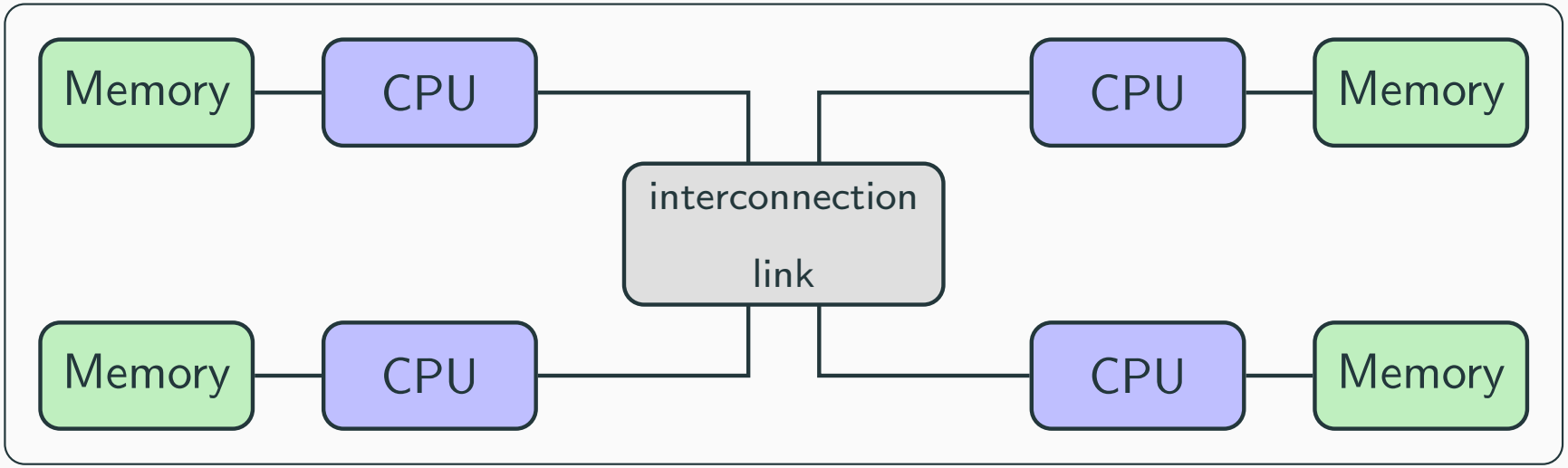
Table of contents

Préambule

Table of contents

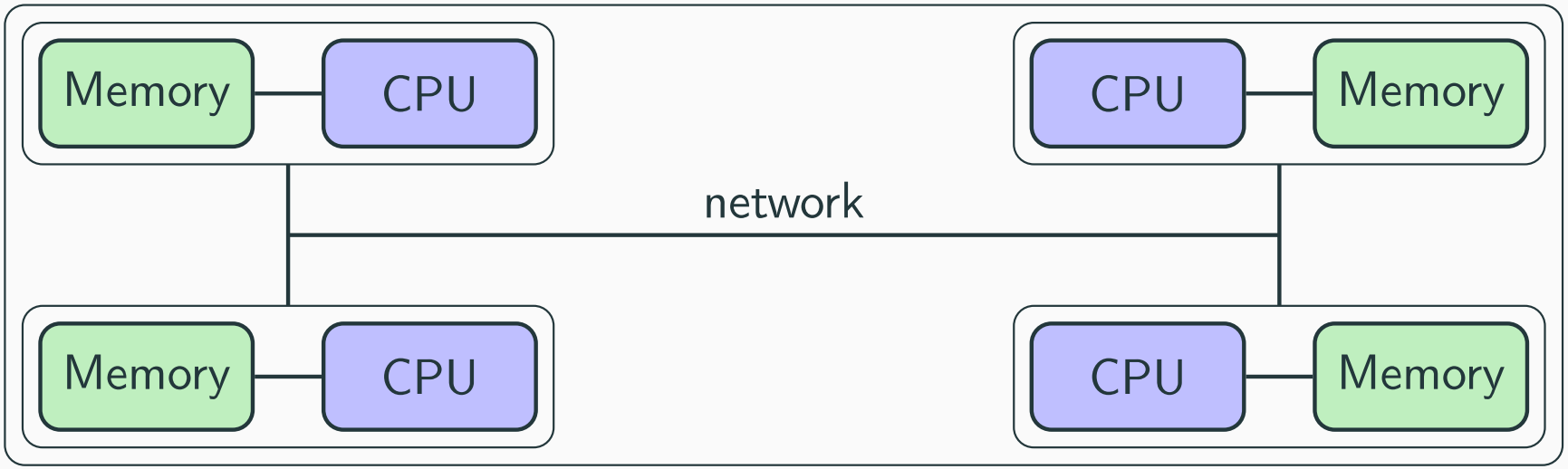
Le calcul parallèle

- Pour offrir une plus grande puissance de calcul, les machines possèdent de plus en plus d'unités de calcul.
- Ces machines peuvent être à architecture à mémoire partagée :



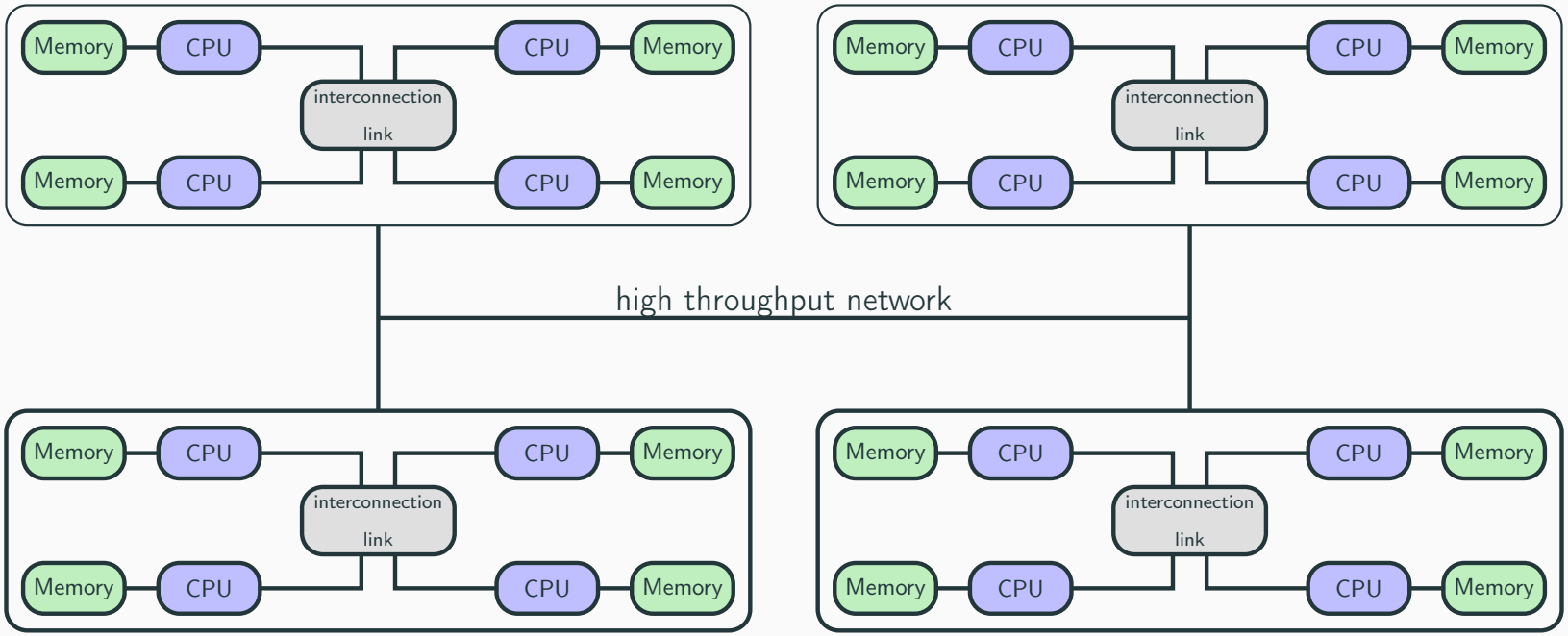
Le calcul parallèle

- Pour offrir une plus grande puissance de calcul, les machines possèdent de plus en plus d'unités de calcul.
- Ces machines peuvent être à architecture à mémoire distribuée :



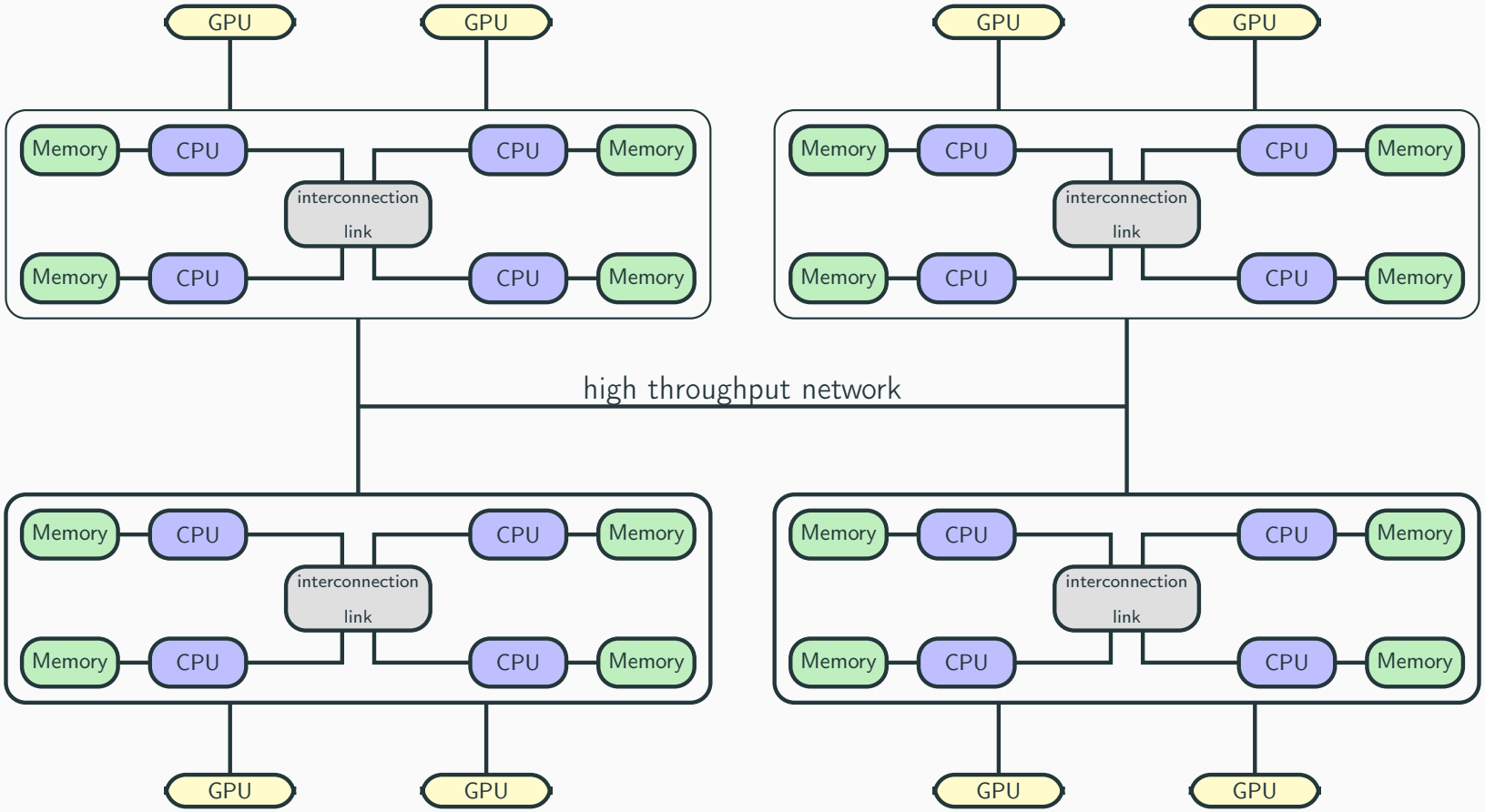
Le calcul parallèle

- Pour offrir une plus grande puissance de calcul, les machines possèdent de plus en plus d'unités de calcul.
- Ces machines peuvent être à architecture à mémoire mixte :



Le calcul parallèle

- Pour offrir une plus grande puissance de calcul, les machines possèdent de plus en plus d'unités de calcul.
- Ces machines peuvent être à architecture mixte à mémoire mixte :



Le calcul parallèle

- Pour tirer parti de ces machines, il faut utiliser des modèles de programmation parallèle.

- Les modèles de programmation parallèle permettent l'exécution **simultanée** de séquences d'instructions sur des processeurs et/ou cœurs différents.

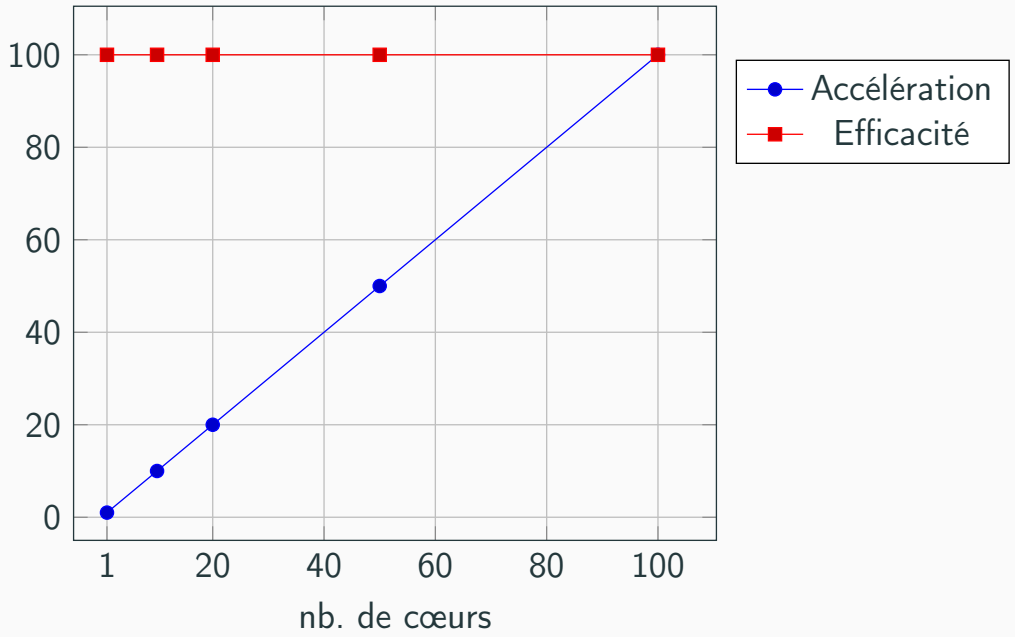
Accélération et efficacité

- L'**accélération** (*speed-up*) A_n et l'**efficacité** E_n sont des mesures de la qualité de la parallélisation.
- Soient t_s le temps d'exécution séquentiel et t_n le temps d'exécution parallèle sur n processeurs, l'accélération et l'efficacité s'écrivent :
  - $A_n = \frac{t_s}{t_n}$
  - $E_n = \frac{A_n}{n}$

Accélération et efficacité

- L'accélération et l'efficacité parfaites valent :

$$t_n = \frac{t_s}{n} \qquad \Rightarrow \qquad A_n = n \quad \text{et} \quad E_n = 100\%$$



Extensibilité forte

- La parallélisation permet de diminuer le temps de restitution d'un calcul en le distribuant sur plus d'unités de calcul.

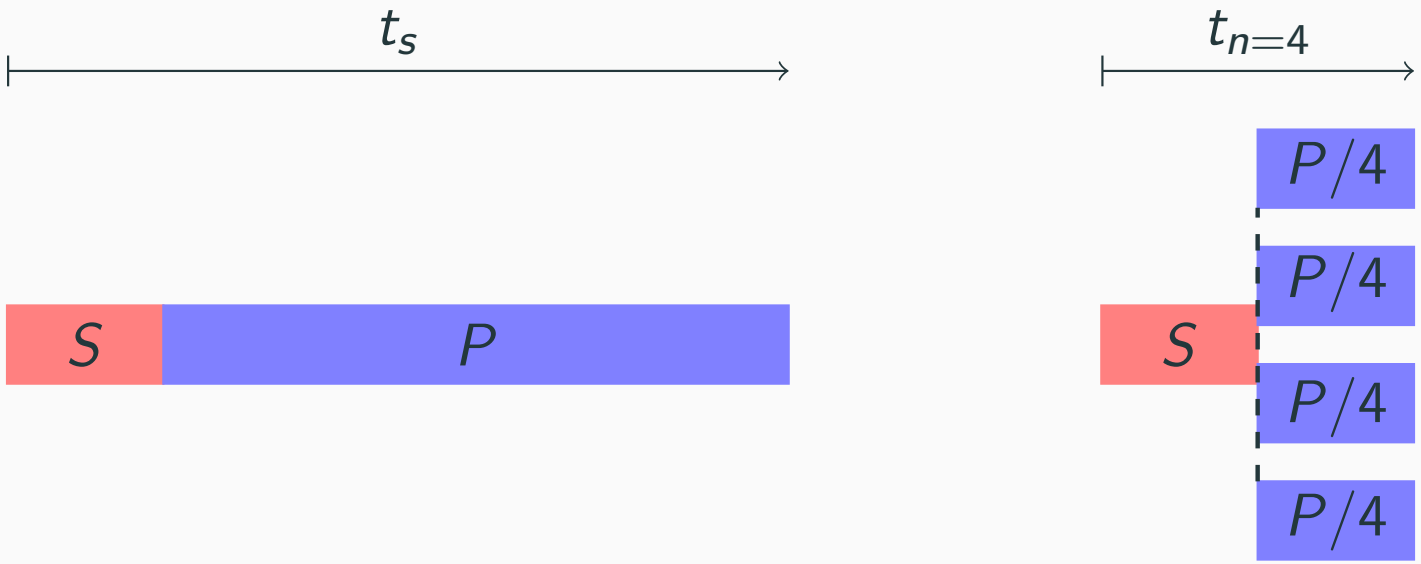
- **Extensibilité forte** : comportement d'un programme parallèle lorque le nombre de processeurs n augmente pour résoudre un problème de taille fixe.

Loi d'Amdahl

Accélération théorique maximale obtenue en parallélisant idéalement un code résolvant un problème de taille fixe :

$$A_n = \frac{t_s}{t_n} = \frac{(S+P)}{(S+P/n)} = \frac{1}{(S+P/n)}$$

- P : fraction parallèle du code
- $S = 1 - P$: fraction séquentielle du code



Loi d'Amdahl

Accélération théorique maximale obtenue en parallélisant idéalement un code résolvant un problème de taille fixe :

$$A_n = \frac{t_s}{t_n} = \frac{(S+P)}{(S+P/n)} = \frac{1}{(S+P/n)}$$

- P : fraction parallèle du code
- $S = 1 - P$: fraction séquentielle du code

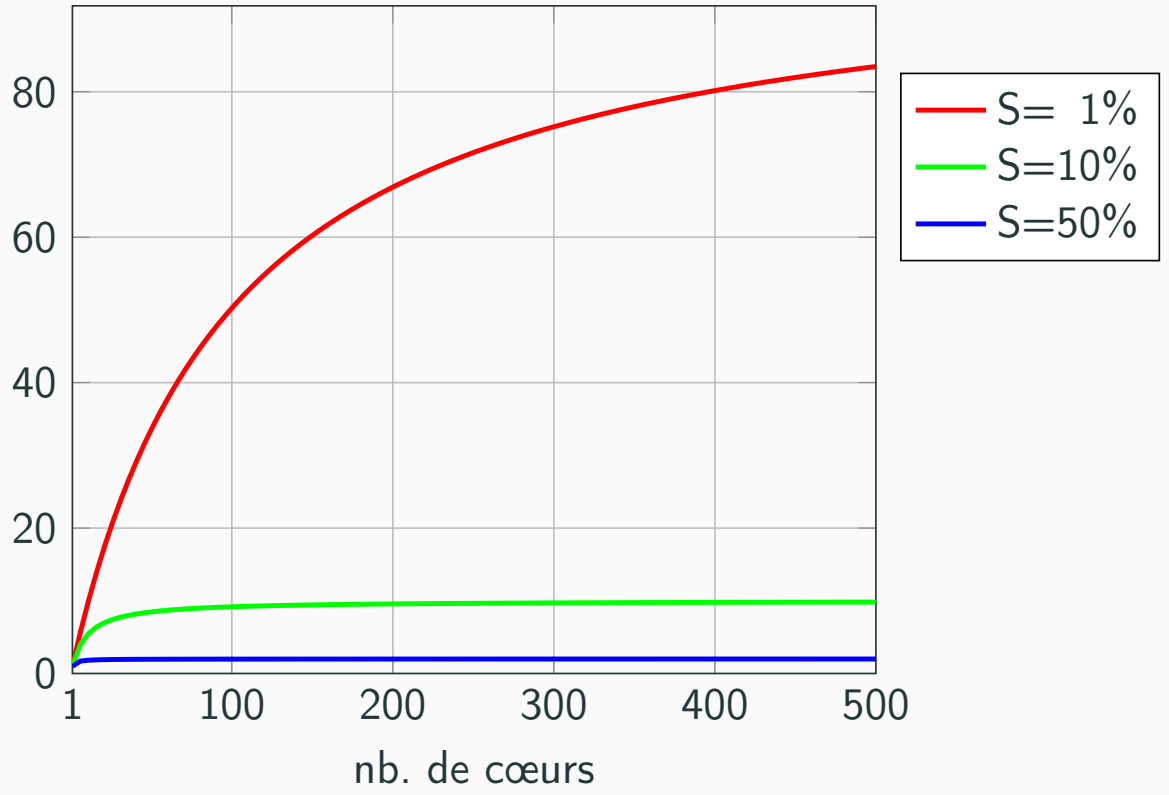
Remarques :

- La loi d'Amdahl décrit l'extensibilité forte.
- L'accélération est limitée par la partie non-parallélisée du code.
- Pour n grand on a $A_n \to \frac{1}{S}$

Loi d'Amdahl

Exemples :

- Pour $S = 50\%$, l'accélération maximale vaut 2
- Pour $S = 10\%$, l'accélération maximale vaut 10
- Pour $S = \ \ 1\%$, l'accélération maximale vaut 100



Extensibilité faible

- La parallélisation permet de réaliser des calculs de taille de plus en plus grande en le distribuant sur plus d'unités de calcul.

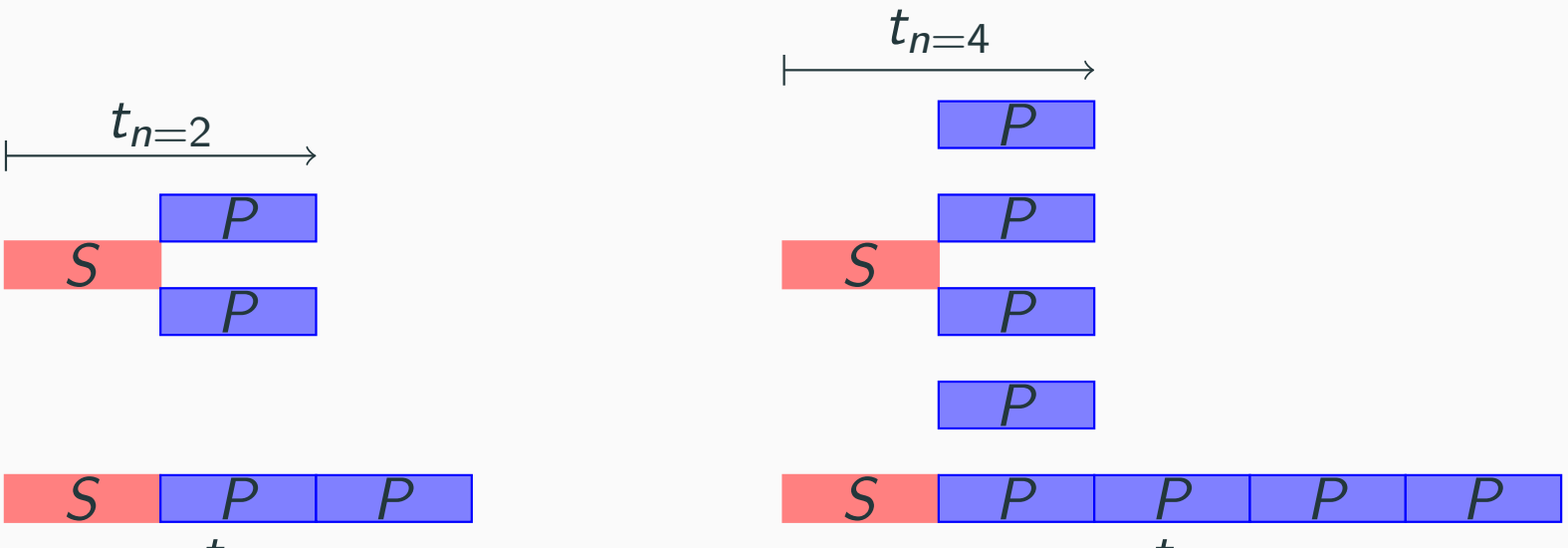
- **Extensibilité faible** : comportement d'un programme parallèle lorque le nombre de processeurs n pour résoudre un problème dont la taille par processeur est fixée.

Loi de Gustafson

Accélération théorique maximale obtenue en parallélisant idéalement un code résolvant un problème de taille fixe par processeur et en supposant que la fraction séquentielle n'augmente pas avec la taille globale du problème.

$$A_n = \frac{t_s}{t_n} = \frac{(S + nP)}{(S + P)} = (1 - P) + nP$$

- P : fraction parallèle du code
- $S = 1 - P$: fraction séquentielle du code



Loi de Gustafson

Accélération théorique maximale obtenue en parallélisant idéalement un code résolvant un problème de taille fixe par processeur et en supposant que la fraction séquentielle n'augmente pas avec la taille globale du problème.

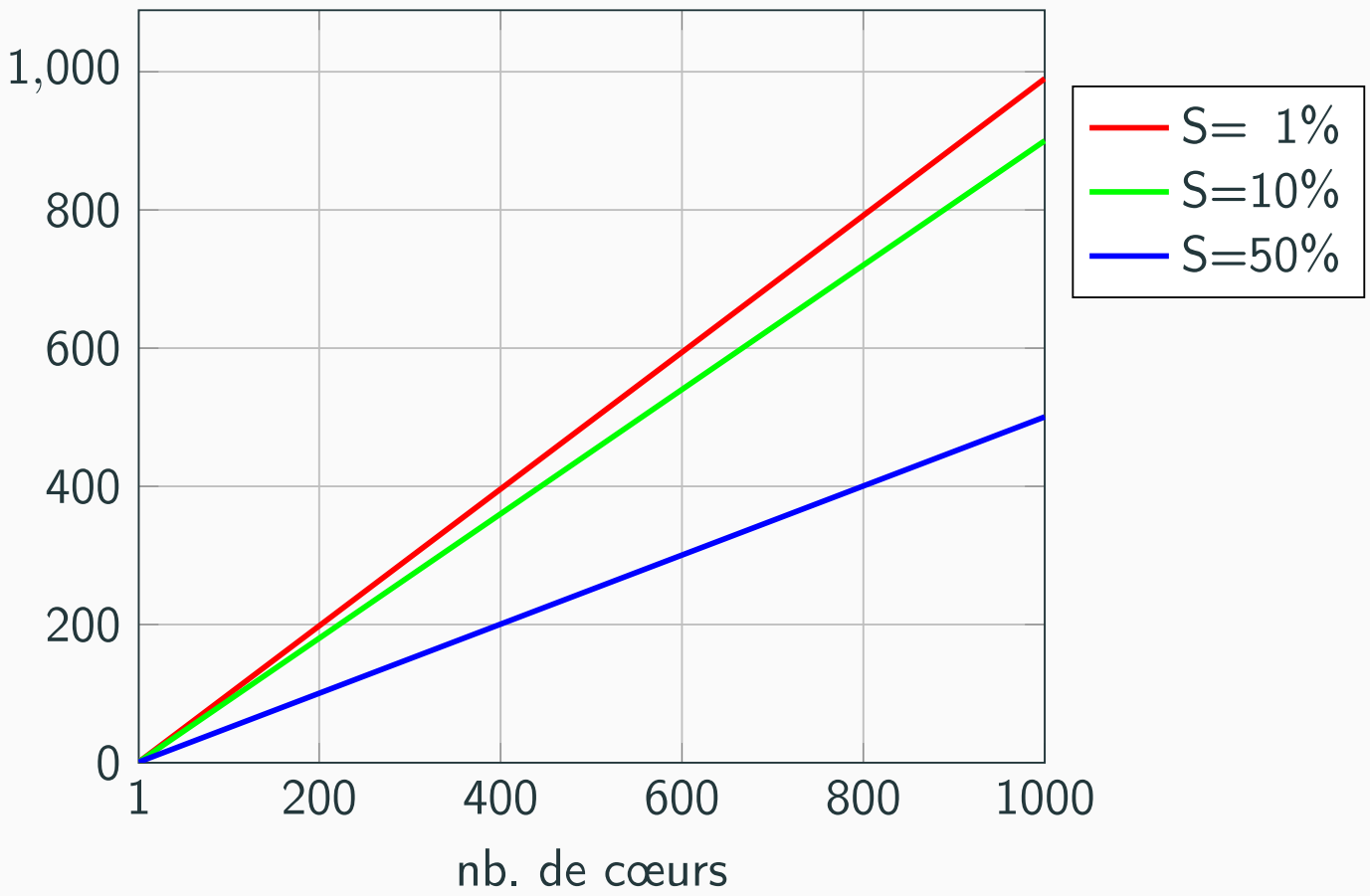
$$A_n = \frac{t_s}{t_n} = \frac{(S + nP)}{(S + P)} = (1 - P) + nP$$

- P : fraction parallèle du code
- $S = 1 - P$: fraction séquentielle du code

Remarques :

- La loi de Gustafson décrit l'extensibilité faible.
- L'accélération est linéaire.

Loi de Gustafson

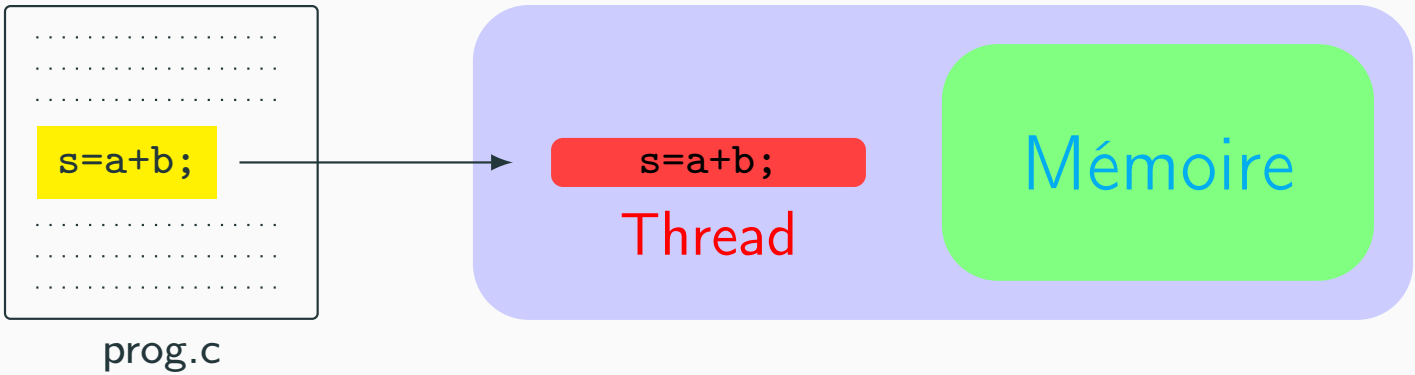


Programmation parallèle à mémoire partagée : OpenMP

Table of contents

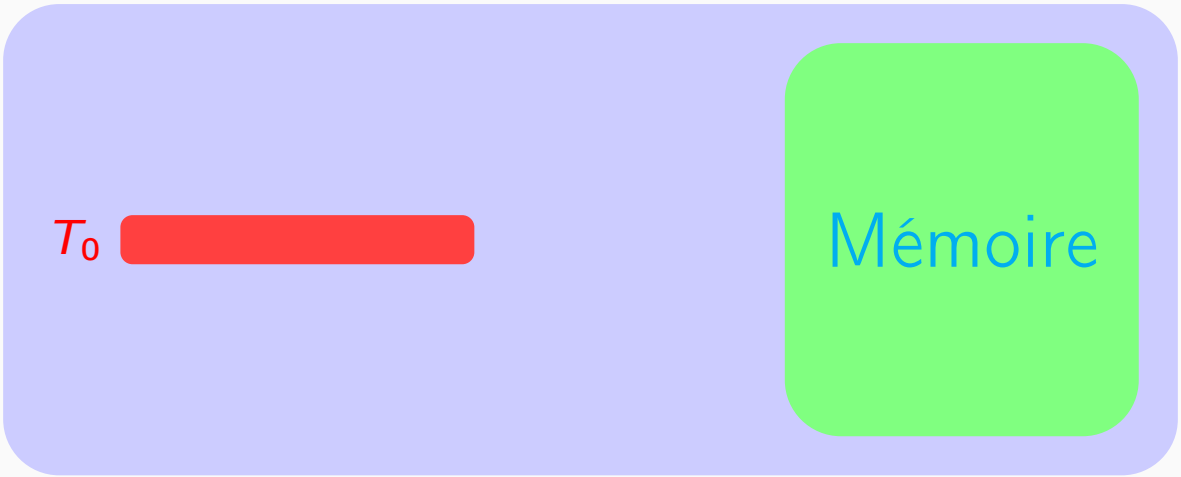
Processus et thread

- Un **processus** est un programme en cours d'exécution auquel est associé un contexte d'exécution.
- Le **contexte d'exécution** est constitué :
  - d'un espace d'adressage permettant au processus d'avoir accès aux instructions et aux données du programme chargées en mémoire ;
  - des fichiers ouverts ;
  - ...
- Un processus contient au moins un **thread** en charge de l'exécution des instructions du programme



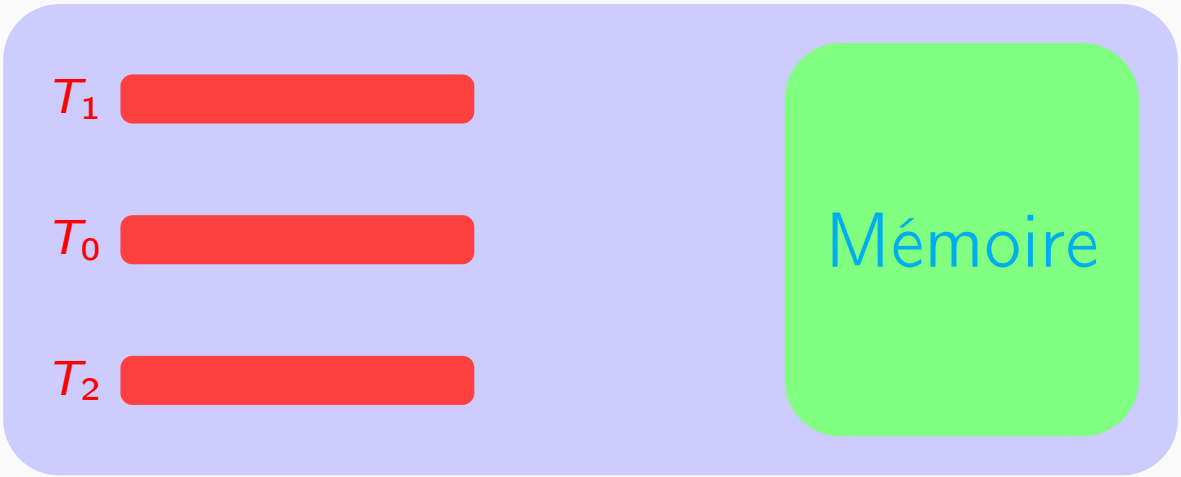
Principes

- Un programme **multi-threads** s'exécute dans un processus unique.



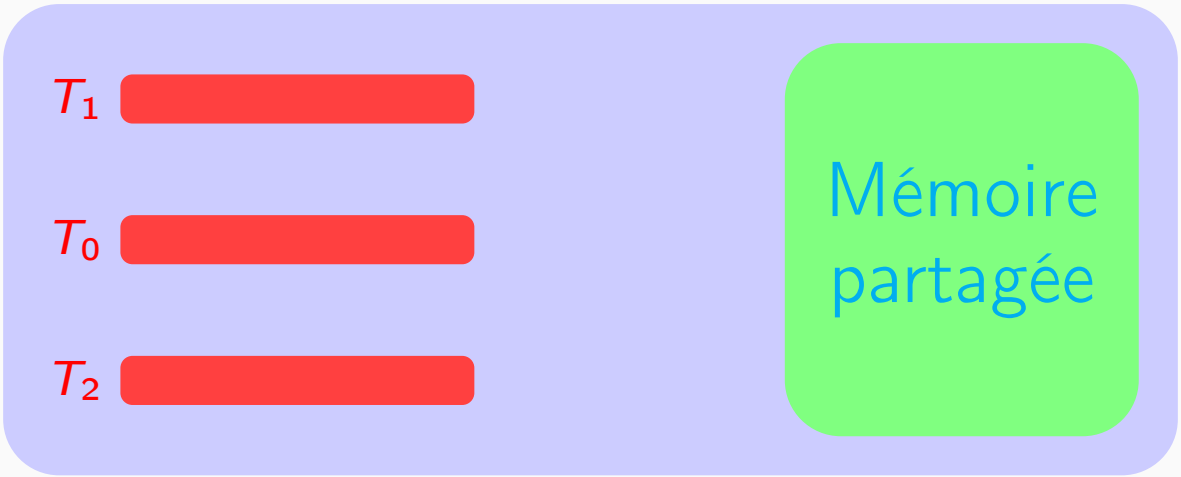
Principes

- Ce processus active plusieurs threads.



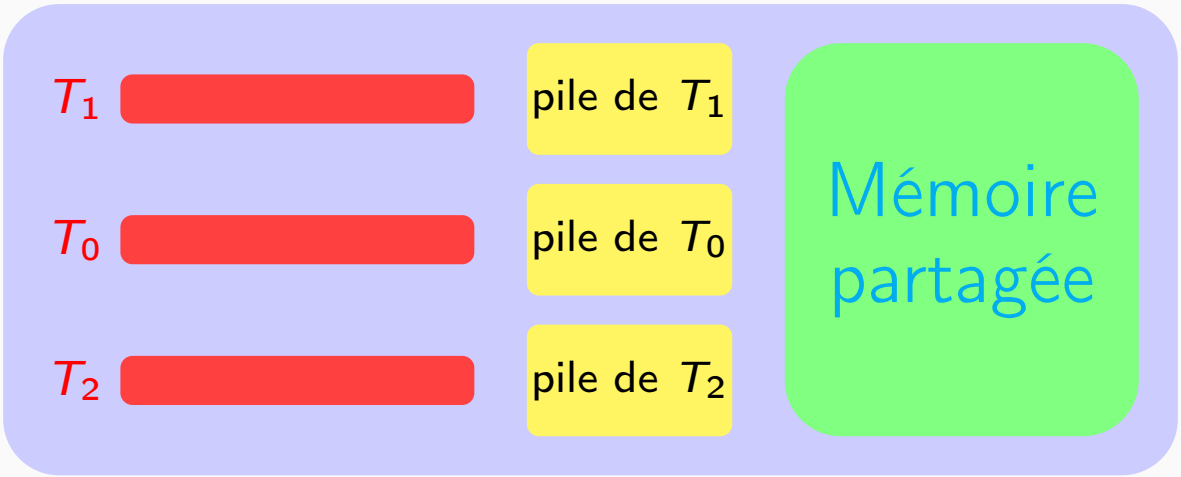
Principes

- Le contexte d'exécution du processus (notamment la mémoire) est partagé par l'ensemble des threads.



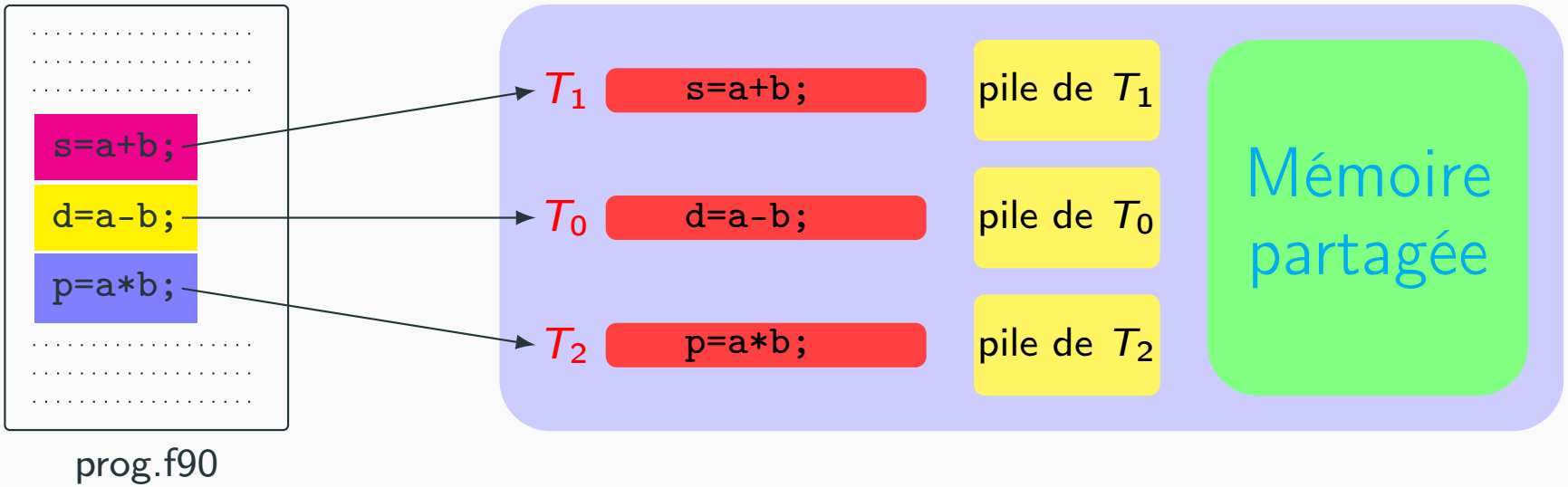
Principes

- Chaque thread dispose d'un espace mémoire privé (pile), invisible des autres threads.



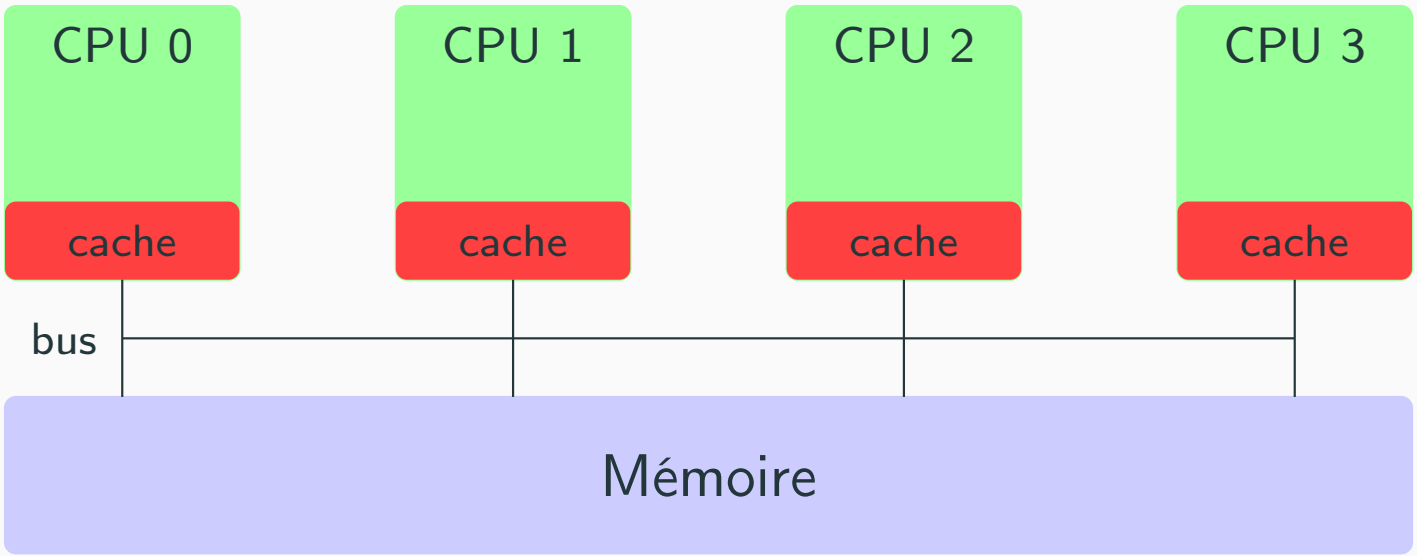
Principes

- Les instructions du programme peuvent être réparties sur les différents threads afin de s'exécuter de manière parallèle.



Architecture cible

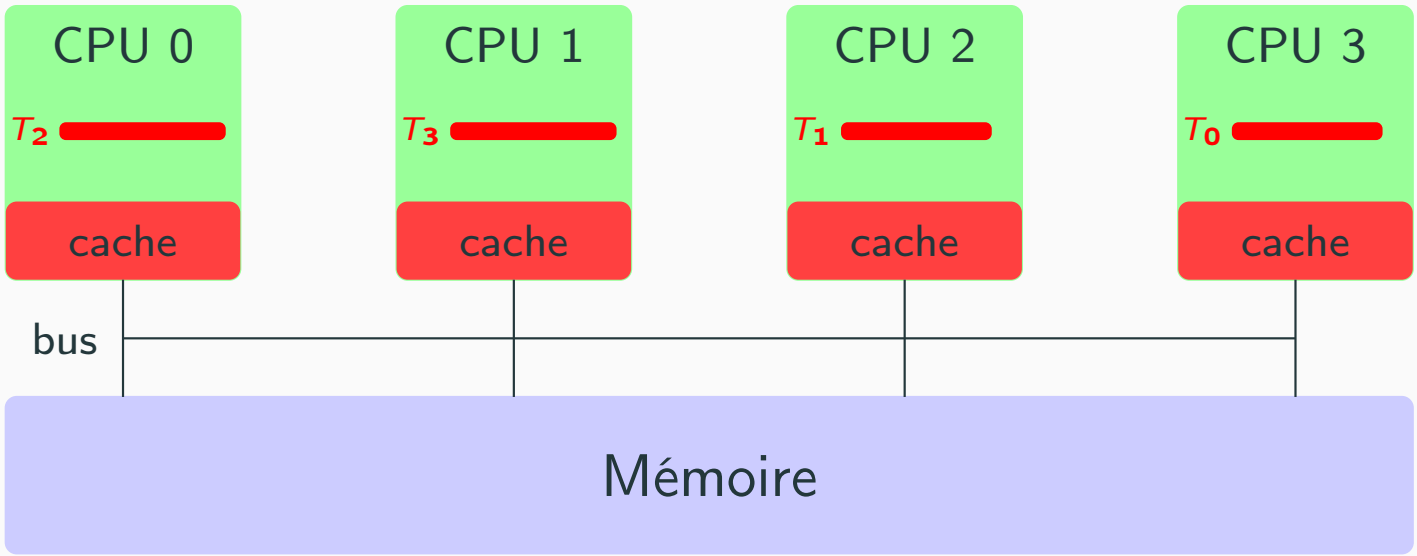
- Pour obtenir une exécution parallèle du programme, les threads doivent être affectés à des processeurs et/ou cœurs différents partageant la même mémoire.
  $\Longrightarrow$ architecture cible : **machine à mémoire partagée**



Machine SMP (Symetric Multiprocessing) de type UMA (Uniform Acess Memory)

Architecture cible

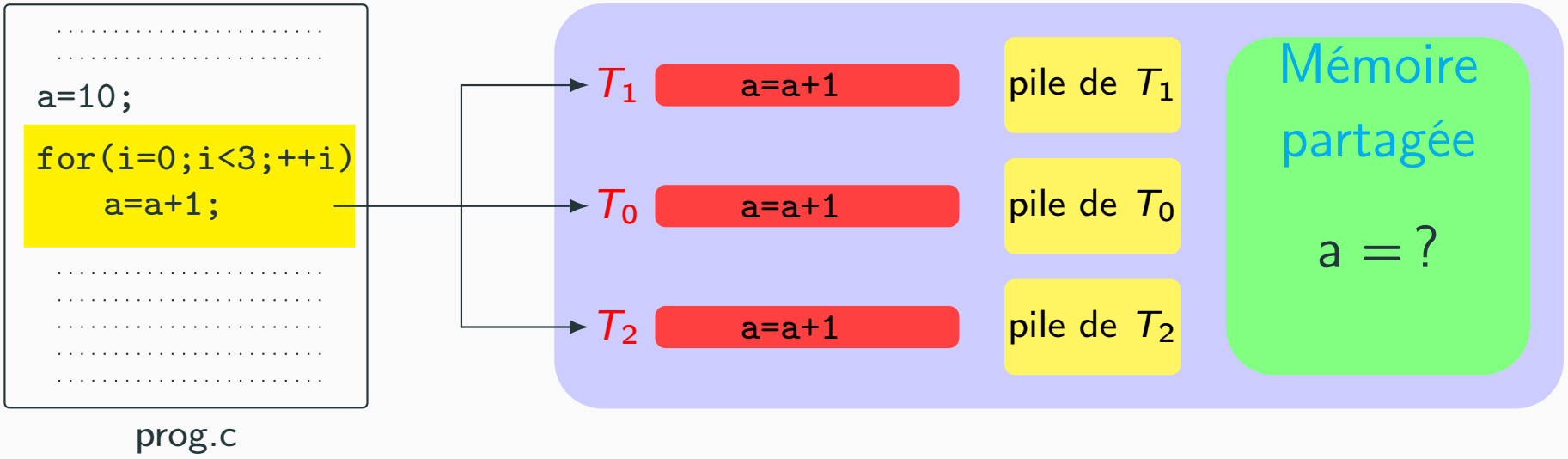
- C'est le système d'exploitation qui distribue les threads sur les différents processeurs ou cœurs de la machine à mémoire partagée.



Machine SMP (Symetric Multiprocessing) de type UMA (Uniform Acess Memory)

Avantage / Inconvenient

- Avantage : l'espace mémoire global, visible de tous les threads, facilite la programmation parallèle.
- Inconvénient : les synchronisations nécessaires lorsque les threads modifient la valeur d'une même variable partagée limite l'extensiblité.



Mise en oeuvre

- Plusieurs bibliothèques permettent de gérer les threads :
  - OpenMP ;
  - Pthreads (C, C++) ;
  - TBB (Intel Threading Building Blocks) (C++) ;
  - Kokkos (C++) ;
  - SYCL (C++)
  - ...

Interface de programmation OpenMP

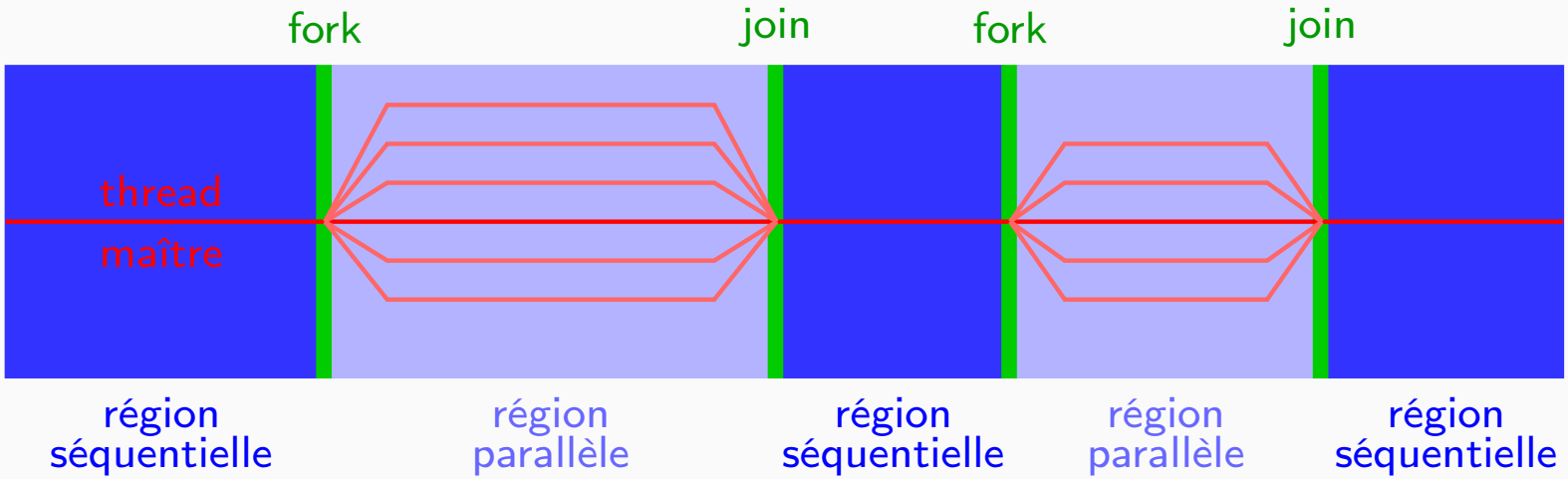
- OpenMP (*Open Multi-Processing*) est une **interface de programmation** (API) pour générer un programme multi-threads sur architecture à mémoire partagée.

- L'interface de programmation fournit :
  - des directives de compilation ;
  - des sous-programmes ;
  - des variables d'environnement.

- Cette API est :
  - disponible pour le Fortran, le C et le C++ ;
  - supportée par de nombreux systèmes et compilateurs (gnu, intel...).

Modèle d'exécution : fork-join

- Un programme OpenMP est une succession de régions séquentielles et parallèles.
- A l'entrée d'une région parallèle, de nouveaux threads sont créés (fork) permettant l'exécution du code en parallèle.
- Ces threads sont désactivés (join) à la fin de la région parallèle, l'éxecution du programme continue en mode séquentiel.
- Le nombre de threads peut être différent d'une région parallèle à l'autre.



Directives

- Les directives OpenMP permettent de :
  - débuter et terminer une région parallèle ;
  - contrôler la répartition du travail ;
  - synchroniser les threads ;
  - ...

- Ce sont des lignes ayant une syntaxe particulière et prises en compte par le compilateur uniquement si l'option permettant leur interprétation est spécifiée (sinon elles sont interprétées comme des commentaires ce qui permet de préserver le code séquentiel).

- Syntaxe
  - Pour le fortran format libre :

```
!$omp directive [clause[[,] clause]...]
```

  - Pour le C/C++ :

```
#pragma omp directive [clause[[,] clause]...]
```

Premier exemple utilisant la directive `parallel`

```c
#include <stdio.h>
int main()
{
  #pragma omp parallel
  {
  printf("Hello World !\n");
  }

  return 0;
}
```

Compilation

- L'interprétation des directives est activée par une option du compilateur :
  - **`-fopenmp`** pour les compilateurs GNU (gcc, gfortran) ;
  - **`-qopenmp`** pour les compilateurs Intel (icc, ifort).
- La variable d'environnement **`OMP_NUM_THREADS`** permet de définir le nombre de threads à lancer.

Exemple de compilation

```c
#include <stdio.h>
int main()
{
  #pragma omp parallel
  {
  printf("Hello World !\n");
  }
  return 0;
}
```

Compilation sans -fopenmp

```
> gcc prog.c
> ./a.out
Hello World !
```

Compilation avec -fopenmp

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=2
> ./a.out
Hello World !
Hello World !
```

Sous-programmes

- Les principaux sous-programmes spécifiques d'OpenMP sont :
  - **`omp_get_num_threads()`** : retourne le nombre de threads dans la région parallèle ;
  - **`omp_get_thread_num()`** : retourne le rang du thread ;
  - **`omp_set_num_threads(num_threads)`** : spécifie le nombre de threads pour les prochaines régions parallèles ;
  - **`omp_in_parallel()`** : retourne 1 (ou `T`) si elle est exécutée dans une région parallèle 0 (ou `F`) sinon
- Les prototypes sont à définir avant utilisation (fichier d'inclusion ou module).

OpenMP : Région parallèle

- La directive **`parallel`** est la directive OpenMP la plus importante.
- Elle permet la création d'une région parallèle, c'est-à-dire qu'elle permet l'activation (*fork*) de $N-1$ threads par le thread maître (thread de rang 0).
- Dans une région parallèle, chaque thread exécute le code qui s'y trouve inclus.
- A la fin de la région parallèle, les $N-1$ threads créés par le thread maître sont désactivés et seul le thread maître continue à s'exécuter (*join*).

Région parallèle

```c
#include <stdio.h>
int main()
{
  printf("Avant region parallele\n");
  #pragma omp parallel
  {
    printf("Pendant region parallele\n");
  }
  printf("Apres region parallele\n");

  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=2
> ./a.out
Avant region parallele
Pendant region parallele
Pendant region parallele
Apres region parallele
```

Définir le nombre de threads

- Le nombre de threads exécutant une région parallèle peut être défini de plusieurs façons :
  - par la variable d'environnement **`OMP_NUM_THREADS`** (avant l'exécution du programme) ;
  - en utilisant la procédure **`omp_set_num_threads`** (pendant l'execution du programme) ;
  - grâce à la clause **`num_threads`** de la directive `parallel` (pendant l'exécution du programme).
- Le nombre de threads peut être choisi indépendamment du nombre de processeurs et/ou cœurs, leur répartition sur ceux-ci est à la charge du système d'exploitation.

Définir le nombre de threads : exemple langage C

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

int main()
{
  int num_threads, thread_num;

  #pragma omp parallel private(num_threads, thread_num)
  {
    num_threads = omp_get_num_threads();
    thread_num = omp_get_thread_num();
    if (thread_num == 0)
    {
      printf("1: nb threads : %d\n", num_threads);
    }
  }

  omp_set_num_threads(4);
  #pragma omp parallel private(num_threads, thread_num)
  {
    num_threads = omp_get_num_threads();
    thread_num = omp_get_thread_num();
    if (thread_num == 0)
    {
      printf("2: nb threads : %d\n", num_threads);
    }
  }

  #pragma omp parallel private(num_threads, thread_num) \
                       num_threads(6)
  {
    num_threads = omp_get_num_threads();
    thread_num = omp_get_thread_num();
    if (thread_num == 0)
    {
      printf("3: nb threads : %d\n", num_threads);
    }
  }
```

```c
  /* suite du  programme */

  #pragma omp parallel private(num_threads, thread_num)
  {
    num_threads = omp_get_num_threads();
    thread_num = omp_get_thread_num();
    if (thread_num == 0)
    {
      printf("4: nb threads : %d\n", num_threads);
    }
  }

  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=2
> ./a.out
1: nb threads : 2
2: nb threads : 4
3: nb threads : 6
4: nb threads : 4
```

Script SLURM pour un job OpenMP

```
$ cat myscript.sh
#! /bin/bash

#SBATCH --job-name=job_omp
#SBATCH --nodes=1 # number of nodes
#SBATCH --ntasks=1  # number of tasks (a single process here)
#SBATCH --cpus-per-task=20  # number of OpenMP threads
#SBATCH --time=00:10:00     # maximum execution time
#SBATCH --output=job_omp.%j.out # name of the standard output file
#SBATCH --error=job_omp.%j.err # name of the error output file
#SBATCH --partition=cpu_shared # partition to use
#SBATCH --account=formation # account

export OMP_NUM_THREADS=20

# launch the program
./my_omp_program

$ sbatch myscript.sh
Submitted batch job 579005
```

Exercice : OpenMP hello world

1. copier le répertoire `/mnt/beegfs/project/formation/2025/omp_hello` dans votre `$WORKDIR`.
2. charger le module gcc
3. compiler les deux programmes `omp_hello_world_01` et `omp_hello_world_02` avec gcc en utilisant le Makefile avec et sans l'option `-fopenmp`
4. soumettre le script SLURM fourni (`job_omp.sh`) pour executer le programme
5. vérifier le résultat en fonction du nombre de threads choisi

Variables partagées et privées

- Au sein d'une région parallèle, une variable est soit **partagée** soit **privée**.

- Une variable partagée :
  - se trouve dans la mémoire du processus : tous les threads accèdent à la même instance de cette variable ;
  - est déclarée par la clause **`shared`** de la directive `parallel`.

- Une variable privée :
  - se trouve dans la pile de chaque thread (duplication) ;
  - est déclarée par la clause **`private`** ou **`firstprivate`** de la directive `parallel`.

- Par défaut, sauf exceptions (indices de boucles des directives `do/for` ...), toute variable d'une région parallèle est partagée.

- La clause **`default(...)`** permet de modifier le statut par défaut dans une région parallèle.

Statut des variables : exemple

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  int a, thread_num;
  a = 100;
  #pragma omp parallel private(thread_num)
  {
  thread_num = omp_get_thread_num();
  printf("a vaut : %d et thread_num vaut: %d\n", a, thread_num);
  }
  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=2
> ./a.out
a vaut : 100 et thread_num vaut : 0
a vaut : 100 et thread_num vaut : 1
```

Valeur par défaut d'une variable privée

- Une variable rendue privée par la clause **`private`** a une valeur indéterminée en entrée de région parallèle :

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  int a, thread_num;
  a = 100;
  #pragma omp parallel default(none) private(a, thread_num)
  {
  thread_num = omp_get_thread_num();
  a = a + thread_num;
  printf("a vaut : %d\n",a);
  }
  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=2
> ./a.out
a vaut : 1
a vaut : 192176864
```

Clause `firstprivate`

- La clause **`firstprivate`** force l'initialisation à la valeur de la variable avant l'entrée dans la région parallèle :

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  int a, thread_num;
  a = 100;
  #pragma omp parallel default(none) private(thread_num) firstprivate(a)
  {
  thread_num = omp_get_thread_num();
  a = a + thread_num;
  printf("a vaut : %d\n",a);
  }
  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=2
> ./a.out
a vaut : 101
a vaut : 100
```

Portée d'une région parallèle

- Une région parallèle s'étend :
  - au code contenu lexicalement dans cette région ;
  - au code des des fonctions appelées.

- Statut des variables d'une fonction appelée dans une région parallèle :
  - Les variables locales sont implicitement privées à chaque thread.
  - Les variables transmises par argument héritent du statut défini au debut de la région parallèle.

Portée d'une région parallèle : exemple

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

void fcn(int x, int *y)
{
   thread_num = omp_get_thread_num();
   *y = x + thread_num();
}

int main()
{
  int a, b;
  a=100;
  #pragma omp parallel shared(a) private(b)
  {
    fcn(a, &b);
    printf("b vaut : %d \n",b);
  }
  return 0;
}
```

```
> gcc -fopenmp prog.c sub.c
> export OMP_NUM_THREADS=2
> a.out
b vaut : 101
b vaut : 100
```

Boucle parallèle

- Une boucle parallèle est une boucle dont les instructions ne présentent pas de dépendances entre les itérations.
- La directive **`for`** en C permet de répartir les itérations d'une boucle entre les threads.
- Cette répartition ne s'applique qu'à la boucle suivant immédiatement la directive **`for`**.
- Les indices d'une boucle suivant la directive **`for`** sont privés par défaut.
- La clause **`schedule`** permet de définir le mode de répartition des itérations.
- Par défaut, une synchronisation globale est effectuée en fin de construction (sauf si la clause **`nowait`** a été spécifiée).

Boucle parallèle : exemple langage C

```c
#include <stdio.h>
#include <omp.h>
int main()
{
const int n = 10;
int thread_num, i;
#pragma omp parallel private(thread_num)
{
  thread_num = omp_get_thread_num();
  #pragma omp for
  for (i=0; i<n; ++i)
  {
  printf("L'iteration %d s'execute sur \
          le thread : %d\n", i, thread_num);
  }
}

return 0;
}
```

```
> gfortran -fopenmp prog.f90
> export OMP_NUM_THREADS=4
> ./a.out
L'iteration  0 s'execute sur le thread :  0
L'iteration  1 s'execute sur le thread :  0
L'iteration  2 s'execute sur le thread :  1
L'iteration  3 s'execute sur le thread :  1
L'iteration  4 s'execute sur le thread :  2
L'iteration  5 s'execute sur le thread :  2
L'iteration  6 s'execute sur le thread :  3
L'iteration  7 s'execute sur le thread :  3
L'iteration  8 s'execute sur le thread :  0
L'iteration  9 s'execute sur le thread :  0
```

Boucle parallèle : répartition des itérations

- Clause `schedule(static,chunk)` :
  les itérations sont attribuées aux threads de manière cyclique par bloc de taille **`chunk`** (à l'exception du dernier bloc dont la taille peut être inférieure).
- Clause `schedule(dynamic,chunk)`
  les itérations sont attribuées aux threads par bloc de taille **`chunk`**.
  Dès qu'un thread a fini de traiter ses itérations, un nouveau bloc lui est attribué.
- Clause `schedule(guided,chunk)`
  les itérations sont attribuées aux threads par bloc de taille décroissante.
  La taille des blocs ne peut être inférieure **`chunk`**. Dès qu'un thread a fini de traiter ses itérations, un nouveau bloc lui est attribué.
- Clause `schedule(runtime)`
  la répartition des itérations est décidée à l'exécution en fonction de la valeur de la variable d'environnement **`OMP_SCHEDULE`**.

Réduction

- La clause **`reduction`** permet de réaliser une opération associative sur une variable partagée.
- Chaque thread réalise le résultat partiel de l'opération de réduction de manière indépendante.
- Le résultat final est alors obtenu lors de la synchronisation des threads.
- Les variables de réduction doivent être partagées dans la section parallèle associée.

Réduction : exemple

```c
#include <stdio.h>
int main()
{
  int const n = 100;
  double x[n], y[n], xy;
  int i;

  for (i=0;i<n;++i)
  {
    x[i]=sqrt((i+1));
    y[i]=sqrt((i+1));
  }

  #pragma omp parallel for reduction(+:xy)
  for (i=0; i<n; ++i)
  {
    xy += x[i] * y[i];
  }
  printf("Produit scalaire de x par y : \%f\n", xy);
  printf("Resultats attendus : \%f\n", (n*(n+1))/2.);

  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=6
> ./a.out
Produit scalaire de x par y : 5050.000000
Resultats attendus : 5050.000000
```

Sections parallèles

- La directive **`sections`** permet de répartir l'exécution de parties de code indépendantes sur différents threads.
- Les parties de code à repartir sont repérées par la directive **`section`** au sein de la construction **`sections`**.
- Par défaut, une synchronisation globale est effectuée en fin de construction (sauf si la clause **`nowait`** a été spécifiée).

Sections parallèles : exemple

```
int main()
{
  const int n = 20;
  double x[n], y[n], xy;
  int thread_nb, i;
  xy = 0.;
  #pragma omp parallel private(thread_nb)
  {
    thread_nb = omp_get_thread_num();
    #pragma omp sections
    {
    #pragma omp section
    {
    printf("Initialisation du vecteur x \
                    thread %d\n", thread_nb);
    for (i=0;i<n;++i) x[i]=sqrt((i+1));
    }
    #pragma omp section
    {
    printf("Initialisation du vecteur y \
                    thread %d\n", thread_nb);
    for (i=0; i<n; ++i) y[i] = sqrt(i+1.);
    }
    }
```

```
    #pragma omp for reduction(+:xy)
    for (i=0; i<n; ++i)
    {
      xy += x[i] * y[i];
    }
  }
  printf("Produit scalaire de x par y : \
           %f\n", xy);
  printf("Résultats attendus :  \
          %f\n", (n*(n+1))/2.);

  return 0;
}
```

```
> gcc -fopenmp prog.c -lm
> export OMP_NUM_THREADS=4
> a.out
Initialisation du vecteur y thread :      2
Initialisation du vecteur x thread :      1
Produit scalaire de x par y :   210.00000000
Resultats attendus :   210.00000000
```

Exécution exclusive

- Les directives **`master`** et **`single`** permettent de faire exécuter une partie de code uniquement par un thread.
- La directive **`master`** impose qu'il s'agit du thread maître.
- En fin de construction, la directive :
  - **`master`** ne contient pas de synchronisation implicite.
  - **`single`** contient une synchronisation implicite.

Exécution exclusive : exemple

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  int a, thread_num;
  #pragma omp parallel private(thread_num)
  {
    thread_num  = omp_get_thread_num();
    #pragma omp single
    {
      printf("Entrer un entier : ");
      scanf("%d", &a);
    }
    printf("Thread %d : l'entier entre est %d\n", thread_num, a);
  }
  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=4
> ./a.out
Entrer un entier : 3
Thread 0 : l'entier entre est 3
Thread 1 : l'entier entre est 3
Thread 2 : l'entier entre est 3
Thread 3 : l'entier entre est 3
```

Synchronisations

- La directive **`barrier`** permet de synchroniser l'ensemble des threads dans une région parallèle.
- La directive **`critical`** permet d'imposer qu'une partie de programme soit exécutée par un seul thread à la fois.
- La directive **`atomic`** permet d'imposer qu'une instruction soit exécutée par un seul thread à la fois.
  - L'instruction doit avoir une forme particulière
  - Son effet est valable uniquement sur l'instruction suivant la directive.

Synchronisations : exemple

```c
#include <stdio.h>
#include <omp.h>
int main()
{
  int a, thread_num;
  #pragma omp parallel private(thread_num)
  {
    thread_num  = omp_get_thread_num();
    #pragma omp master
    {
      printf("Entrer un entier : ");
      scanf("%d", &a);
    }
    #pragma omp barrier
    printf("Thread %d : l'entier entre est %d\n", thread_num, a);
  }
  return 0;
}
```

```
> gcc -fopenmp prog.c
> export OMP_NUM_THREADS=4
> ./a.out
Entrer un entier : 3
Thread 0 : l'entier entre est 3
Thread 1 : l'entier entre est 3
Thread 2 : l'entier entre est 3
Thread 3 : l'entier entre est 3
```

Coût des directives

- Exemple* pour 12 threads (Intel Xeon CPU E5-2670 v3 @ 2.30GHz)

Directive	Surcoût (μs)
`parallel`	2.98
`for`	1.95
`barrier`	1.46
`single`	2.41
`critical`	0.31
`reduction`	4.24
`atomic`	0.09

Table 1: overheads (μs) à comparer au temps de cycle ($0.43 ns$)

*EPCC OpenMP Microbenchmarks, http ://www2.epcc.ed.ac.uk/

Parallélisation conditionnelle

- La clause **`if`** permet de mettre en place une parallélisation conditionnelle :

```
#include <stdio.h>
int main()
{
  const int n = 10000;

  #pragma omp parallel if(n > 1000)
  {
  /* Partie de code executee sur         */
  /* l'ensemble des threads si n > 1000 */
  }
  return 0;
}
```

Exemple : calcul de Pi

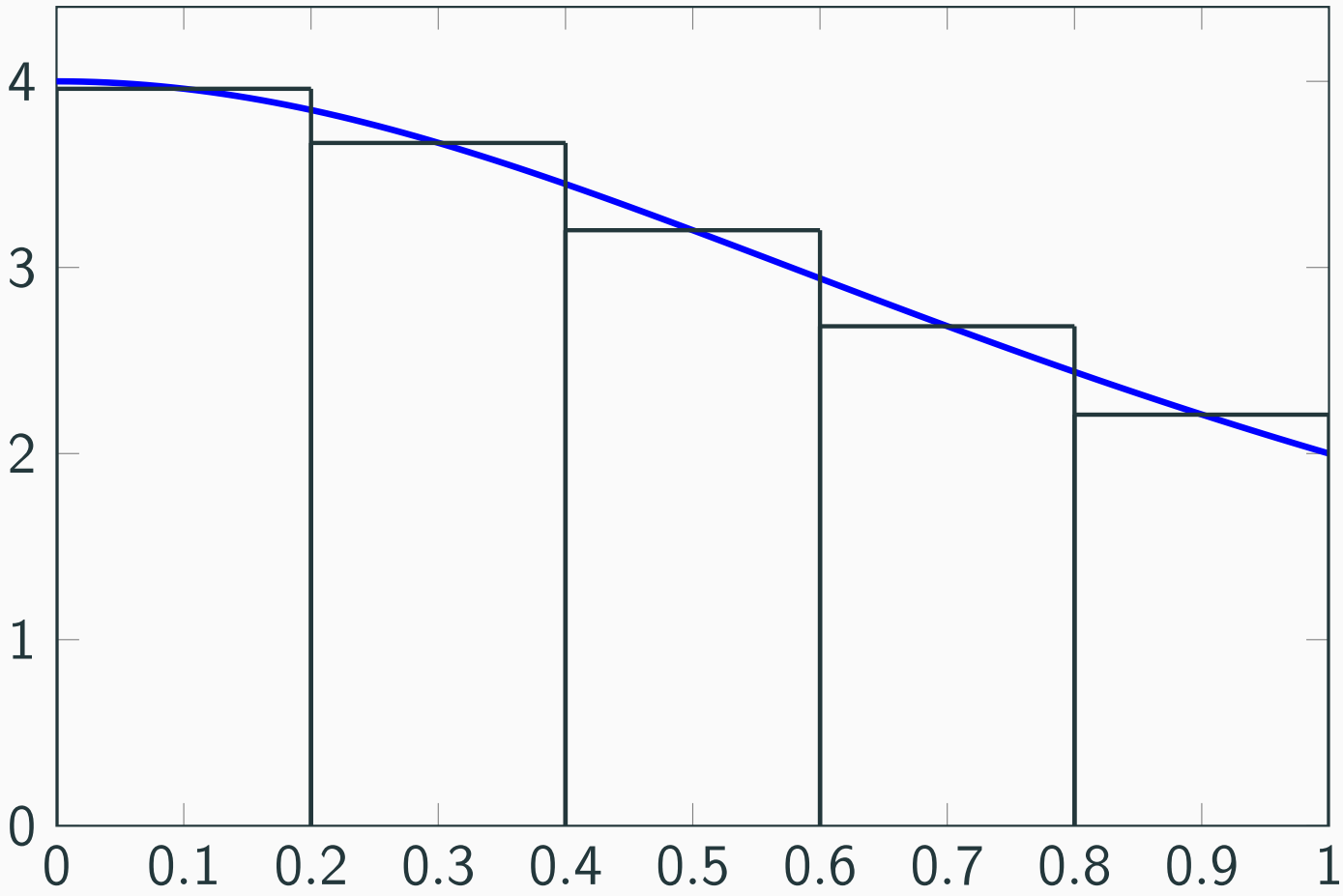
π peut être calculé simplement par intégration :

$$\pi = \int_0^1 f(x)dx \text{ avec } f(x) = \frac{4}{1+x^2}$$

Une approximation de cette relation est :

$$\pi \simeq h \sum_{i=1}^{n} f(x_{i-1/2}) \text{ avec } h = \frac{1}{n} \text{ et } x_{i-1/2} = \frac{i-1/2}{n}$$

Exemple : calcul de Pi



Exercice : calcul de Pi

Programme séquentiel

```c
#include <stdio.h>

int main (int argc, char *argv[])
{
  const int n = 3000000;
  double h, x, pi, sum;
  int i;

  h = 1.0 / ((double)n);

  sum = 0;
  for (i=0; i<n; ++i)
  {
    x = ((double)i - 0.5) * h;
    sum += (4.0 / (1.0 + (x*x)));
  }

  pi = h * sum;

  printf("pi = %lf\n", pi);

  return 0;
}
```

Exercice : calcul de Pi

1. copier le répertoire `/mnt/beegfs/project/formation/2025/omp_pi` dans votre `$WORKDIR`.
2. charger le module intel compiler
3. écrire une version parallèle du programme `pi.c` avec OpenMP
4. compiler et tester les performances du code avec différent nombre de threads

Exercice : calcul de Pi

Solution

```c
#include <stdio.h>

int main (int argc, char *argv[])
{
  const int n = 3000000;
  double h, x, pi, sum;
  int i;

  h = 1.0 / ((double)n);

  sum = 0;
  #pragma omp parallel for private(x) reduction(+:sum)
  for (i=0; i<n; ++i)
  {
    x = ((double)i - 0.5) * h;
    sum += (4.0 / (1.0 + (x*x)));
  }

  pi = h * sum;

  printf("pi = %lf\n", pi);

  return 0;
}
```

Exercise : Product matrix vector

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,j} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,j} & \dots & a_{2,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \dots & a_{i,j} & \dots & a_{i,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,j} & \dots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,j}x_j + \cdots + a_{1,n}x_n \\ a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,j}x_j + \cdots + a_{1,n}x_n \\ \vdots \\ a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,j}x_j + \cdots + a_{i,n}x_n \\ \vdots \\ a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,j}x_j + \cdots + a_{1,n}x_n \end{pmatrix}$$

Exercise : Product matrix vector

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,j} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,j} & \dots & a_{2,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \dots & a_{i,j} & \dots & a_{i,n} \\ \vdots & \vdots & & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,j} & \dots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,j}x_j + \cdots + a_{1,n}x_n \\ a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,j}x_j + \cdots + a_{1,n}x_n \\ \vdots \\ a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,j}x_j + \cdots + a_{i,n}x_n \\ \vdots \\ a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,j}x_j + \cdots + a_{1,n}x_n \end{pmatrix}$$

Exercise : Product matrix vector

1. copy the directory `/mnt/beegfs/project/formation/2025/omp_mxv`
2. load intel compiler and intel mkl module
3. compile the program using the Makefile (`make`)
4. launch job with a matrix size `n = 80000`
5. during execution, connect to the node used by your job (`ssh node0xx`)
6. type `lscpu` to display CPU architecture information
7. load `htop` module
8. type `htop` to display how the threads are distributed on cores

Références

- Site officiel : http ://openmp.org
  Spécifications officielles d'OpenMP
- *OpenMP. Parallélisation multitâches pour machines à mémoire partagée.* J. Chergui, P.-F. Lavallée, Cours Idris, V 22.03, 2020

Programmation parallèle à mémoire distribuée : MPI

Table of contents

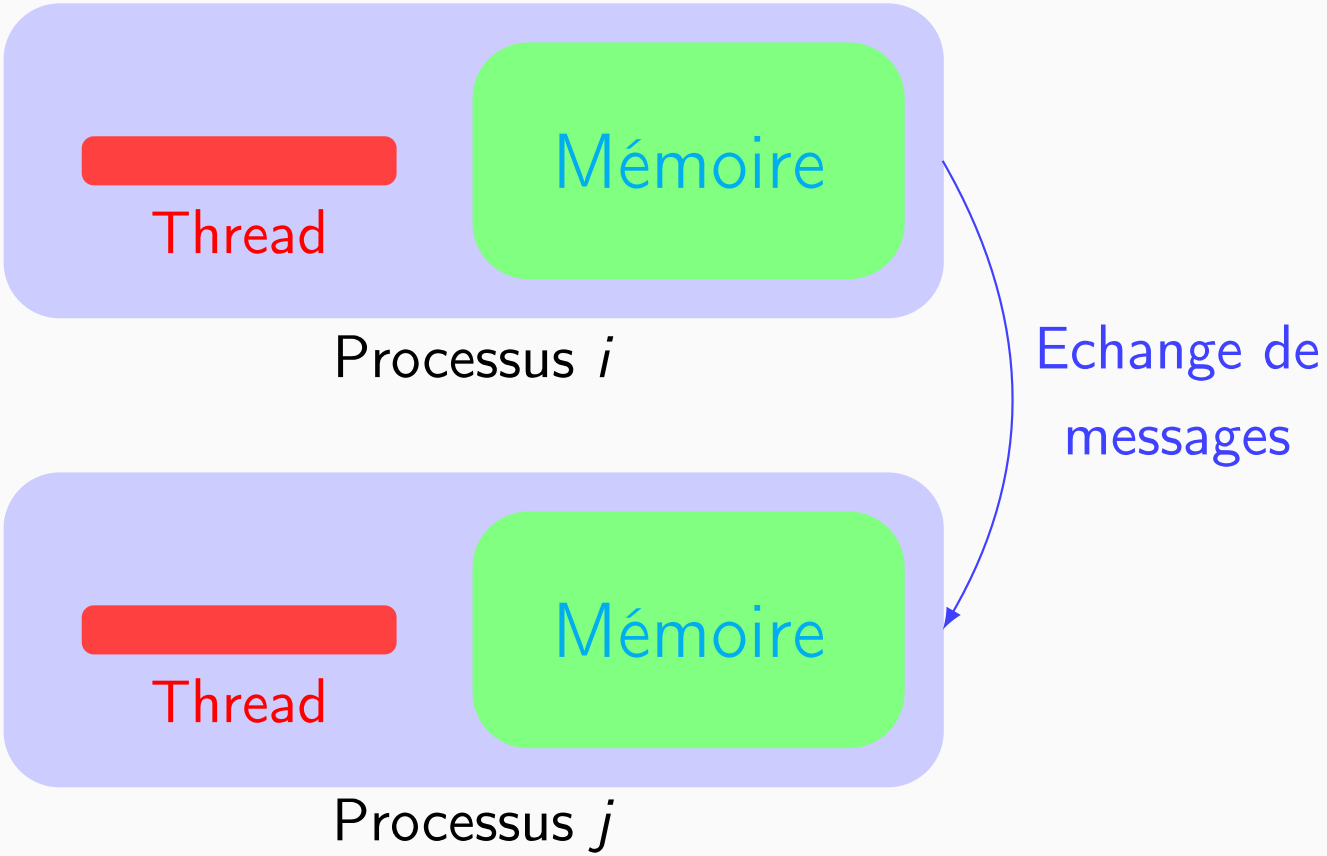
Programmation à mémoire distribuée : échange de messages

Pour ce type de modèle, on utilise le plus souvent le modèle de programmation par échanges de messages :

- Chaque processus exécute un programme sur des données différentes.
- Ce programme peut-être soit :
  - le même pour chaque processus (modèle d'exécution SPMD pour Single Program Multiple Data)
  - différent entre chaque processus (modèle d'exécution MPMD pour Multiple Program Multiple Data)
- Chaque processus dispose de ses propres données, sans accès direct à celles des autres.
- Les processus peuvent s'échanger des messages entre eux pour :
  - transférer des données
  - se synchroniser

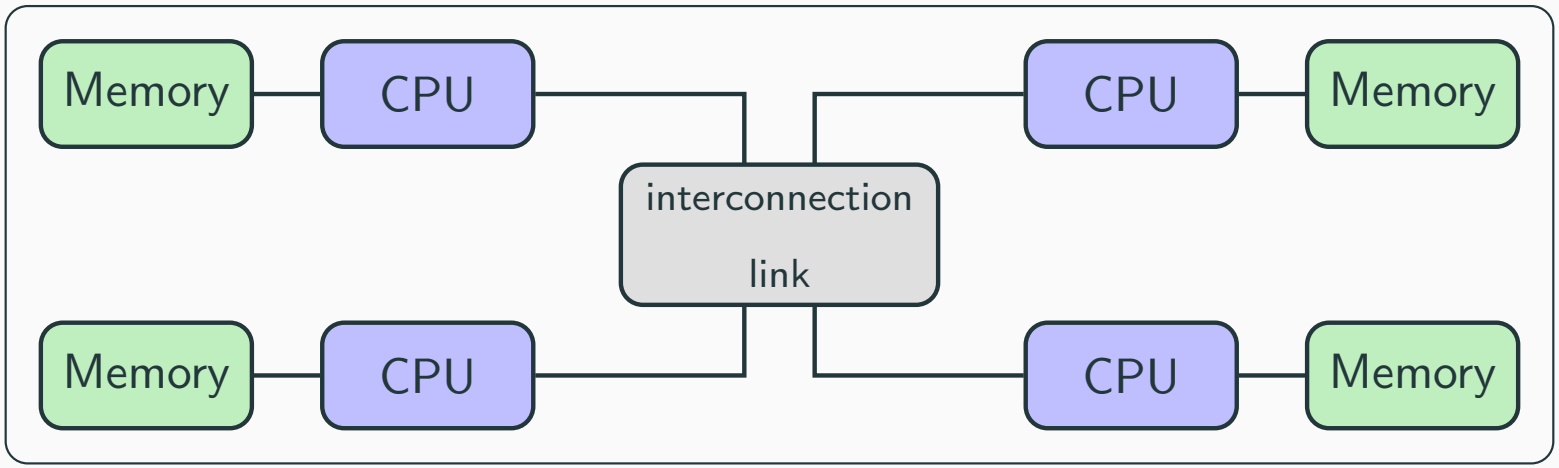
Concepts d'échange de messages

- Un processus source i envoie des données au processus j.
- Le processus cible j reçoit des données du processus i.



Architecture cible

- Ce type de modèle permet d'utiliser les machines à memoire distribuée.
- Les données échangées entre les processus transitent via le réseau d'interconnexion.



- Ce type de modèle permet d'utiliser les machines à memoire partagée. Dans ce cas, les données sont échangées par copie de blocs de mémoire.

Programmation à mémoire partagée : MPI

MPI (*Message Passing Interface*) est une **interface de programmation** (API) pour générer un **programme par échange de messages**.

- L'interface de programmation fournit des fonctions permettant de gérer :
  - un environnement de d'exécution ;
  - les communication point à point ;
  - les communication collectives ;
  - des groupes de processus ...
- Il existe plusieurs implémentations de cette API :
  - développées par les constructeurs pour leur plate-formes ;
  - disponible en open source (OpenMPI, Mpich2, ...) ;
- La bibliothèque est disponible en Fortran, le C et le C++.

MPI : Compilation et exécution

- Dans la plupart des cas, il faut utiliser les commandes de compilation suivantes :
  - **`mpif77, mpif90`** pour le fortran
  - **`mpicc`** pour le langage C
  - **`mpiCC, mpicxx, mpic++`** pour le langage C++

- Les exécutables doivent en général être lancés par un utilitaire particulier. Celui-ci est dépendant de la machine et de la bibliothèque MPI utilisée (srun, mpirun, prun, ...)

MPI : Environnement

- Un environnement d'exécution est nécessaire pour les programmes MPI :
  - la fonction **`MPI_Init`** permet de l'initialiser ;
  - la fonction **`MPI_Finalize()`** permet de le détruire.
- L'initialisation de l'environnement permet :
  - la création du communicateur **`MPI_COMM_WORLD`** à l'intérieur duquel les échanges entre processus s'effectue
  - l'attribution à chaque processus d'un rang unique dans ce communicateur.
- La fonction **`MPI_Comm_size()`** permet de connaître le nombre de processus dans un communicateur.
- La fonction **`MPI_Comm_rank()`** permet de connaître le rang d'un processus dans un communicateur.

MPI : Premier exemple en C

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

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

  int nb_proc;
  int proc_nb;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nb_proc);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_nb);

  printf("Processus de rang %d parmi %d processus MPI\n", proc_nb, nb_proc);

  MPI_Finalize();

  return 0;
}
```

```
> mpicc -o mpi_simple mpi_simple.c
> mpirun -np 4 ./mpi_simple
Processus de rang 0 parmi 4 processus MPI
Processus de rang 1 parmi 4 processus MPI
Processus de rang 3 parmi 4 processus MPI
Processus de rang 2 parmi 4 processus MPI
```

Script SLURM pour un job OpenMP

```
$ cat myscript.sh
#! /bin/bash

#SBATCH --job-name=job_mpi
#SBATCH --ntasks=4  # number of mpi processes
#SBATCH --ntasks-per-node=2         # number of processes by node
#SBATCH --time=00:10:00     # maximum execution time
#SBATCH --output=job_mpi.%j.out # name of the standard output file
#SBATCH --error=job_mpi.%j.err # name of the error output file
#SBATCH --partition=cpu_dist # partition to use
#SBATCH --account=formation # account

# launch the program
mpilaunch -np 4 ./my_mpi_program

$ sbatch myscript.sh
Submitted batch job 579005
```

Exercise : MPI Hello world

1. copy the directory `/mnt/beegfs/project/formation/mpi_hello/2025`
2. load gcc module and openmpi module (4.1.4)
3. compile the program using the Makefile (`make`)
4. test the program by setting the number of processes in `job.sh` and launching job

MPI : Communications point à point

- Il s'agit de l'envoi de données par un processus émetteur identifié vers un processus récepteur unique.
- Lors de l'appel aux fonctions de communication point à point, il faut préciser :
  - le rang du processus émetteur
  - le rang du processus récepteur
  - le communicateur concerné
  - la longueur et le type des données transmises
  - une étiquette identifiant la communication (tag)
- La fonction **`MPI_Send(...)`** permet à un processus d'envoyer des données à un autre processsus
- La fonction **`MPI_Recv(...)`** permet à un processus de recevoir des données d'un autre processsus

MPI : Communications point à point (exemple langage C)

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

int main(int argc, char **argv)
{
  int proc_nb, tag=100;
  double val_send, val_recv;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_nb);

  val_send = 20. + (double)proc_nb;

  if (proc_nb == 1) MPI_Send(&val_send, 1, MPI_DOUBLE, 3, tag, MPI_COMM_WORLD);

  if (proc_nb == 3)
  {
   MPI_Recv(&val_recv, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
   printf("Le processus de rang %d a recu le reel %lf du processus %d\n",proc_nb,val_recv,1);
  }
  MPI_Finalize();

  return 0;
}
```

```
> mpicc -o mpi_comm_simple mpi_comm_simple.c
> mpirun -np 5 ./mpi_comm_simple
Le processus de rang 3 a recu le reel 21.000000 du processus 1
```

MPI : Communications point à point (deadlock)

- La fonction **`MPI_Send(...)`** est bloquante, c'est à dire qu'elle ne rend la main que lorsque l'envoi a bien été effectué (vers un buffer ou vers le destinataire).
- L'échange de données entre 2 processus peut donc amener à une situation bloquante.
- En effet, considérons l'échange d'une valeur entre les processus de rang 0 et 1. Si pour le processus 0, on écrit :

```
MPI_Send(&val_send, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
MPI_Recv(&val_recv, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
```

et pour le processus 1, on écrit :

```
MPI_Send(&val_send, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
MPI_Recv(&val_recv, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
```

on se trouve dans une situation bloquante si la fonction **`MPI_Send(...)`** ne rend la main que lorsque l'envoi a bien été effectué vers le destinataire (mode synchrone)

MPI : Communications point à point (deadlock)

Pour éviter cela, il suffit de changer l'ordre des appels dans l'un des deux processus :

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

int main(int argc, char **argv)
{
  int proc_nb, tag=100;
  double val_send, val_recv;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_nb);

  if (proc_nb == 0) {
     MPI_Send(&val_send, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
     MPI_Recv(&val_recv, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &status);
  }
  if (proc_nb == 1) {
     MPI_Recv(&val_recv, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
     MPI_Send(&val_send, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
  }

  printf("Le processus de rang %d a recu la valeur %d\n",proc_nb,val_recv);

  MPI_Finalize();
  return 0;
}
```

Exercise : MPI Ping-pong

1. copy the directory `/mnt/beegfs/project/formation/mpi_ping_pong/2025`
2. load gcc module and openmpi module (4.1.4)
3. complete the program `mpi_ping_pong.c` so that process 0 sends a value to process 1 which sends it back
4. compile it using the Makefile (`make`) and test it via SLURM (`job.sh`)

MPI : Une bibliothèque riche

Il est possible d'écrire tout programme MPI avec :

- Les six fonctions suivantes :
  - `MPI_Init()`
  - `MPI_Finalize()`
  - `MPI_Comm_size()`
  - `MPI_Comm_rank()`
  - `MPI_Recv()`
  - `MPI_Send()`
- les types de données MPI : `MPI_CHAR`, `MPI_INTEGER`, `MPI_DOUBLE`, ...
- le communicateur par défaut : `MPI_COMM_WORLD`

Pour aider le programmeur, MPI offre beaucoup de fonctions pour réaliser :

- des communications collectives ;
- des groupes de processus ; item des topologies de processus ;
- des sous-communicateurs ;
- ...

MPI : les communications collectives

Les communications collectives

- Il s'agit de communications impliquant un ensemble de processus.
- Elles permettent d'effectuee une série de communications point à point en une seule opération.

Un exemple : la réduction

- La réduction est un mode de communications collectives où l'échange des données s'accompagne d'opérations sur celles-ci.
- Les opérateurs doivent être associatives.
- Voici quelques opérateurs existants :
  - `MPI_SUM` : somme des éléments
  - `MPI_PROD` : produit des éléments
  - `MPI_MAX` : recherche du maximum
  - `MPI_MIN` : recherche du minimum
  - et encore beaucoup d'autres, dont des opérations définies par le programmeur

MPI : exemple de communications collectives (la réduction)

Les fonctions `MPI_Reduce` et `MPI_Allreduce`

- Elle permet de faire des opérations de réduction sur des données réparties sur différents processus.
- Dans le cas de la fonction `MPI_Reduce`, le résultat de l'opération de réduction est récupéré dans un seul processus.
- Dans le cas de la fonction `MPI_Alleduce`, le résultat de l'opération de réduction est récupéré dans chaque processus.

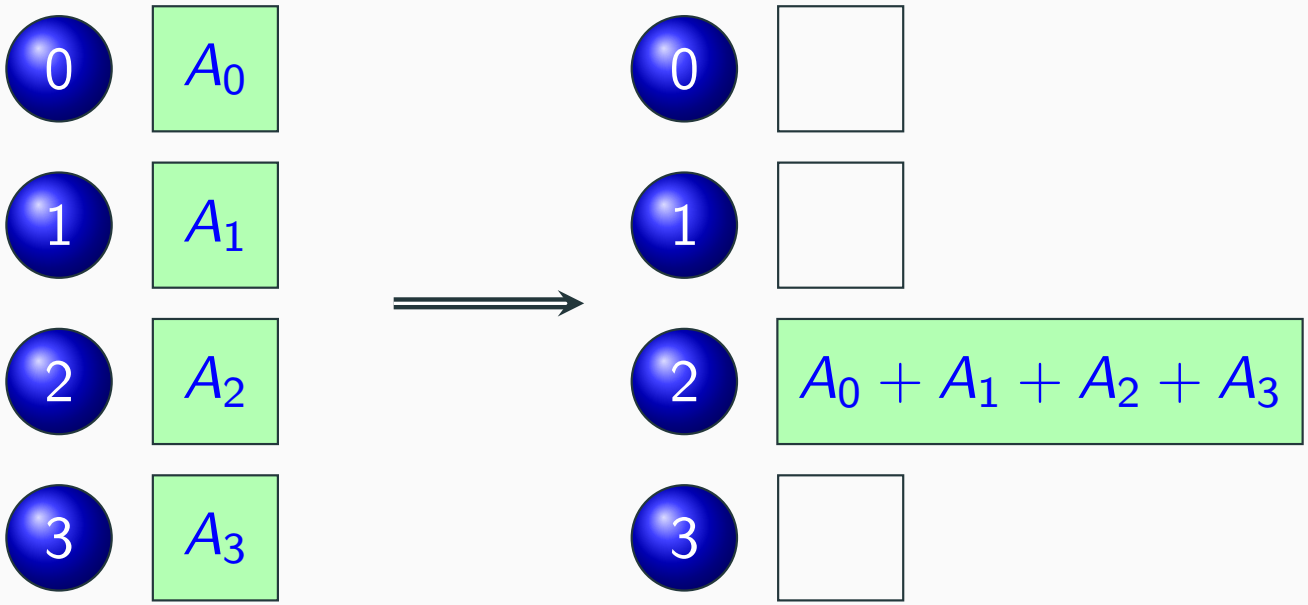


Illustration de la fonction `MPI_Reduce` avec l'opérateur `MPI_SUM`

MPI : exemple de communications collectives (la réduction)

Les fonctions `MPI_Reduce` et `MPI_Allreduce`

- Elle permet de faire des opérations de réduction sur des données réparties sur différents processus.
- Dans le cas de la fonction `MPI_Reduce`, le résultat de l'opération de réduction est récupéré dans un seul processus.
- Dans le cas de la fonction `MPI_Alleduce`, le résultat de l'opération de réduction est récupéré dans chaque processus.

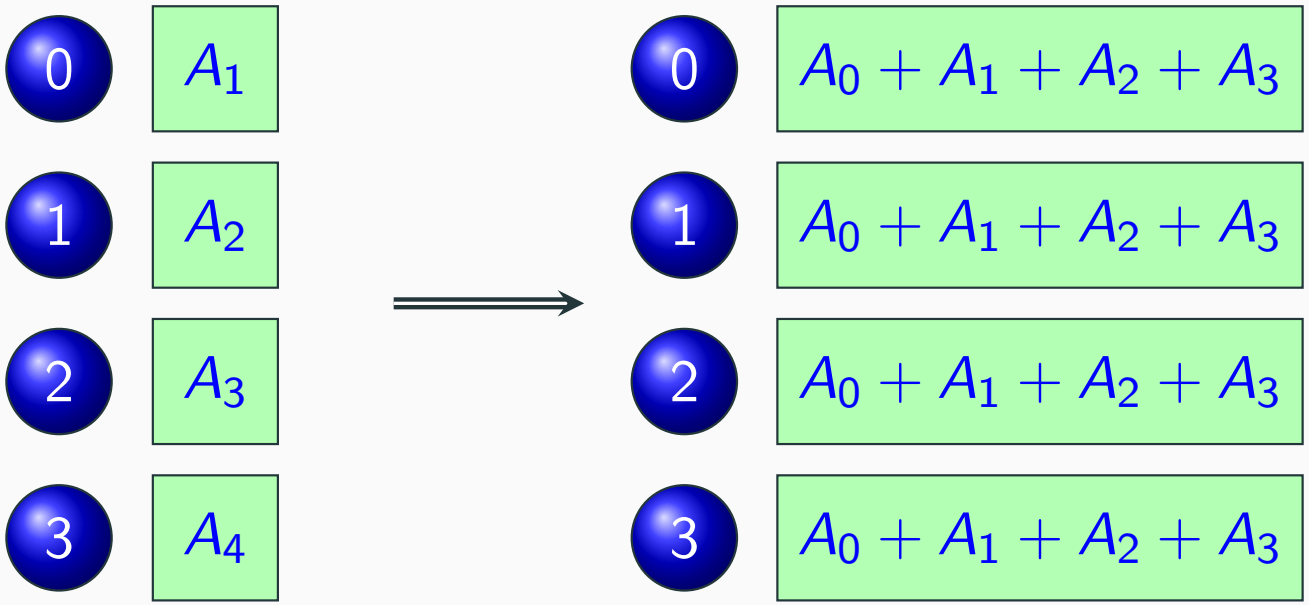


Illustration de la fonction `MPI_Allreduce` avec l'opérateur `MPI_SUM`

MPI : exemple de communications collectives (la réduction)

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

int main(int argc, char **argv)
{
  int proc_nb, sum;
  double val_send, val_recv;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_nb);

  MPI_Reduce(&proc_nb, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if (proc_nb == 0) printf("La somme des rangs des processus vaut %d\n", sum);

  MPI_Allreduce(&proc_nb, &sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  printf("Processus %d : la somme des rangs des processus vaut %d\n", proc_nb, sum);

  MPI_Finalize();
  return 0;
}
```

```
> mpicc -o reduce reduce.c
> mpirun -np 4 reduce
La somme des rangs des processus vaut 6
Processus 0 : la somme des rangs des processus vaut 6
Processus 2 : la somme des rangs des processus vaut 6
Processus 1 : la somme des rangs des processus vaut 6
Processus 3 : la somme des rangs des processus vaut 6
```

MPI : D'autres communications collectives

- synchronisation globale :`MPI_Barrier`
- diffusion générale : `MPI_Bcast`
- diffusion sélective : `MPI_Scatter`
- collecte : `MPI_Gather` et `MPI_Allgather`
- ...

Exemple : calcul de Pi

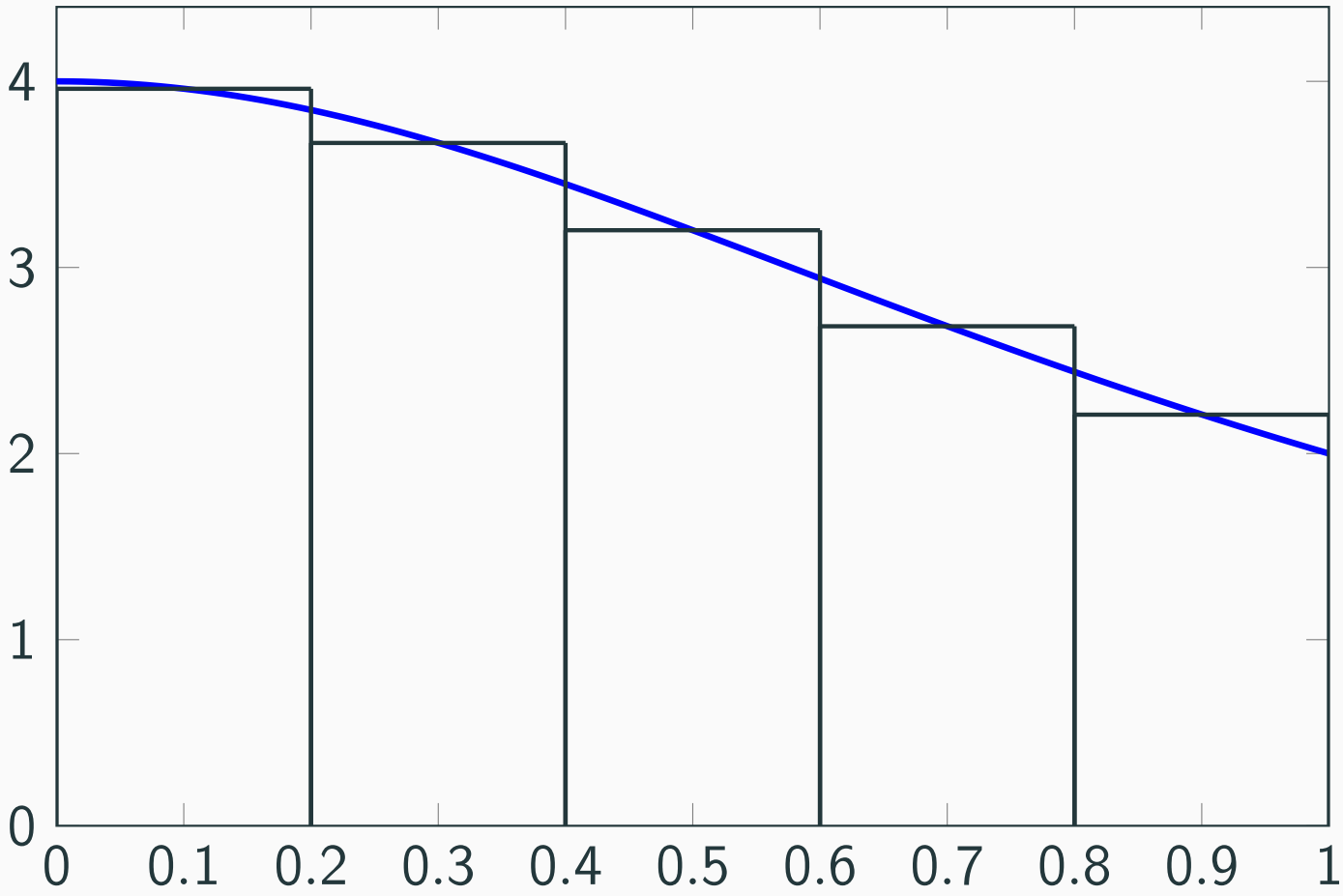
π peut être calculé simplement par intégration :

$$\pi = \int_0^1 f(x)dx \text{ avec } f(x) = \frac{4}{1+x^2}$$

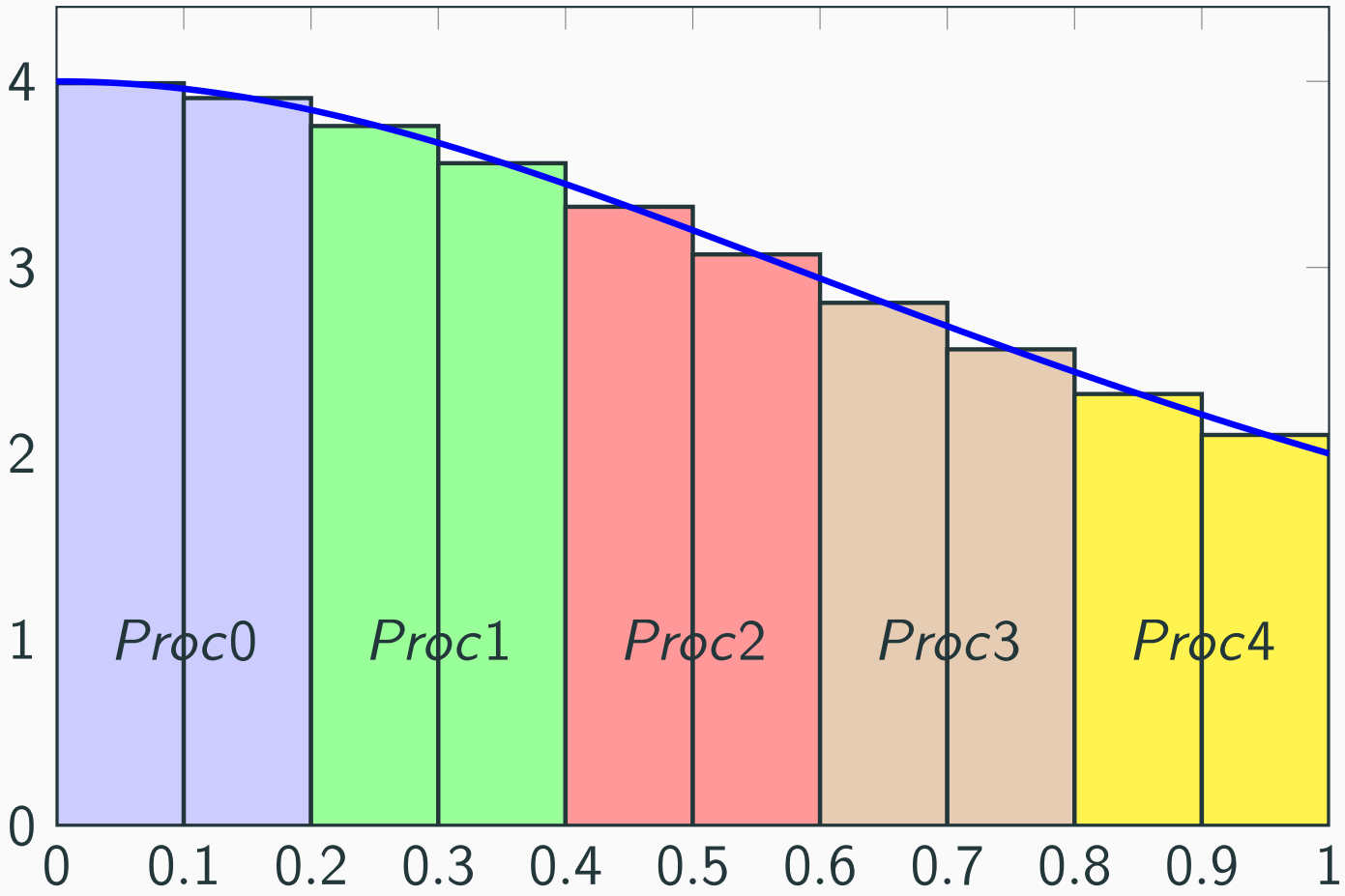
Une approximation de cette relation est :

$$\pi = h\sum_{i=1}^{n} f(x_{i-1/2}) \text{ avec } h = \frac{1}{n} \text{ et } x_{i-1/2} = \frac{i-1/2}{n}$$

Exemple : calcul de Pi



Exemple : calcul de Pi



Exercice : calcul de Pi

1. copier le répertoire `/mnt/beegfs/project/formation/2025/mpi_pi` dans votre `$WORKDIR`.
2. écrire une version parallèle du programme `pi.c` avec MPI
3. compiler et tester avec différent nombre de processus

Exemple : calcul de Pi

Programme MPI

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

int main (int argc, char *argv[])
{
  const int n = 3000000;
  double h, x, pi, sum, loc_sum;
  int i, imin, imax, nloc;
  int proc_nb, nb_proc;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &proc_nb);
  MPI_Comm_size(MPI_COMM_WORLD, &nb_proc);

  h = 1.0 / ((double)n);
  nloc = n / nb_proc;
  imin = proc_nb * nloc + 1;
  imax = (proc_nb + 1) * nloc;

  loc_sum = 0;
  for (i=imin; i<imax; ++i)
  {
    x = ((double)i - 0.5) * h;
    loc_sum += (4.0 / (1.0 + (x*x)));
  }
  MPI_Reduce(&loc_sum, &sum, 1, MPI_DOUBLE, \
                MPI_SUM, 0, MPI_COMM_WORLD);
```

```c
  if (proc_nb == 0)
  {
    pi = h * sum;
    printf("Pi = %lf\n", pi);
  }

  MPI_Finalize();

  return 0;
}
```

Exercice : produit matrice vecteur

1. copier le répertoire `/mnt/beegfs/project/formation/2025/mpi_decomp_mxv` dans votre `$WORKDIR`.
2. compiler et lancer le programme tel quel
3. Comprendre d'où vient l'erreur
4. Trouver une solution et relancer
5. Tester avec différents nombre de processus, sur 1 noeud puis plusieurs
6. Ajouter des directives OpenMP pour calculer le produit matrice vecteur

MPI : Références

- *MPI : A Message-Passing Interface Standard, Version 3.0* Message Passing Interface Forum, High Performance Computing Center Stuttgart (HLRS), 2012.
- *Using MPI, third edition Portable Parallel Programming with the Message-Passing Interface* William Gropp, Ewing Lusk et Anthony Skjellum, MIT Press, 2014.
- *Using Advanced MPI Modern Features of the Message-Passing Interface* William Gropp, Torsten Hoefler, Rajeev Thakur et Erwing Lusk, MIT Press, 2014.
- *Message Passing Interface (MPI)*, Dimitri Lecas, Rémi Lacroix, Serge Van Criekingen, Myriam Peyrounette, Cours Idris, V5.2.21, 2022.

Programmation hybride

- La programmation hybride consiste à utiliser plusieurs modèles de programmation parallèle afin de tirer parti des avantages des différentes approches.
- MPI est utilisé au niveau des processus et un modèle à mémoire partagée comme OpenMP est utilisé à l'intérieur de chaque processus.
- Support des threads dans MPI : les threads peuvent faire des appels MPI
- *Programmation hybride MPI-OpenMP*, Pierre-Francois Lavallée, Philippe Wautelet, Rémi Lacroix, Cours Idris, V3.1.0, 2018.