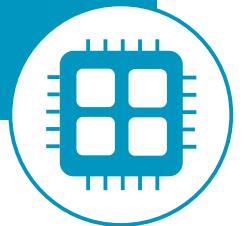


UE Programmation Parallèle

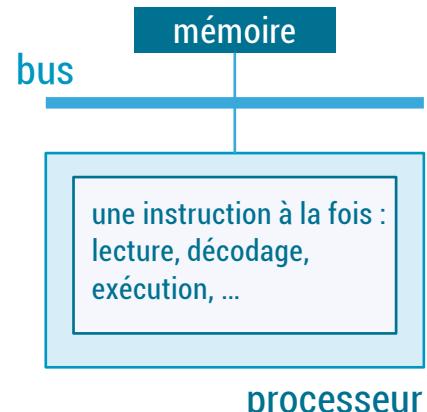
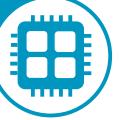
L3 Informatique



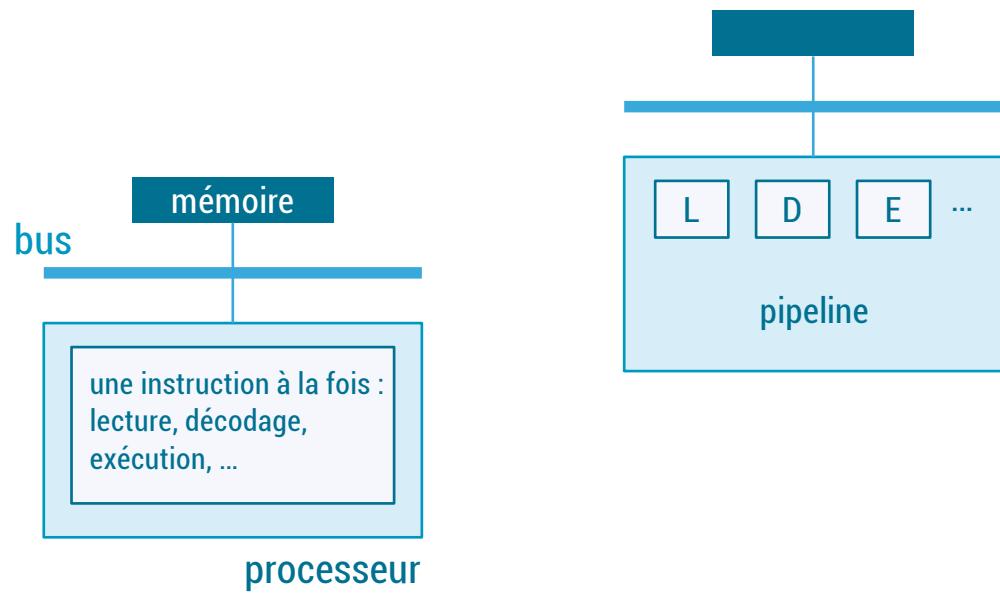
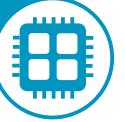
Thomas Carle
Christine Rochange



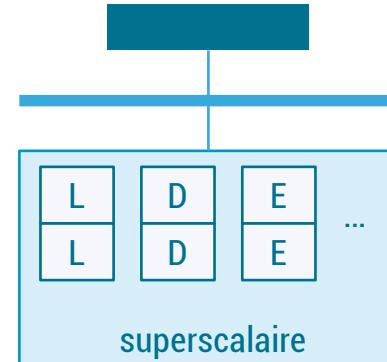
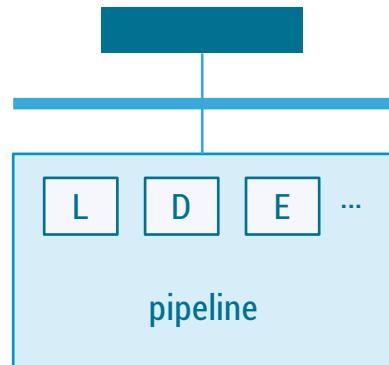
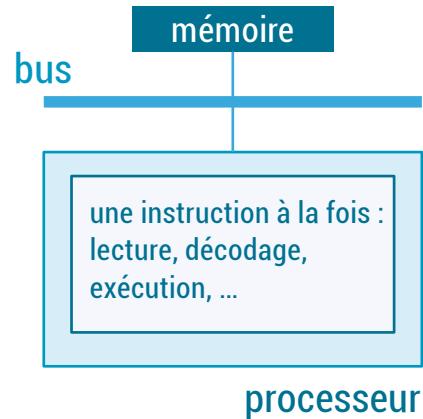
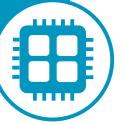
Evolution des processeurs



Evolution des processeurs

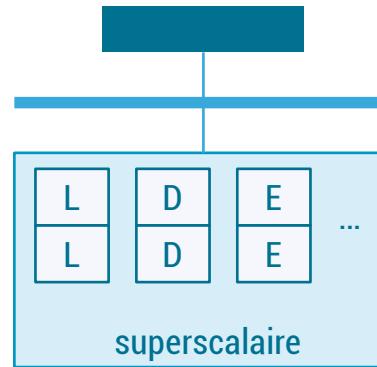
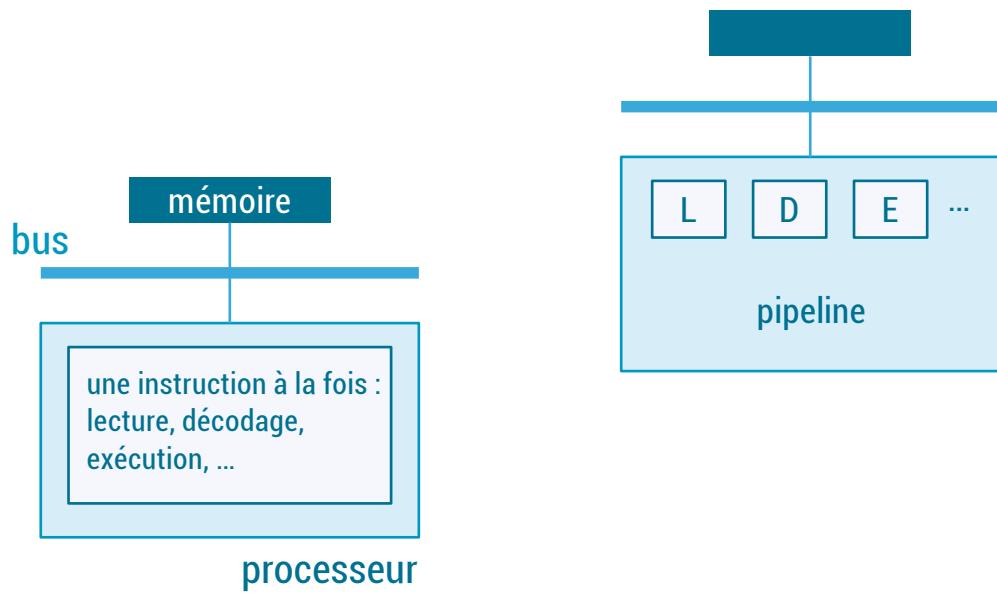
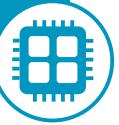


Evolution des processeurs

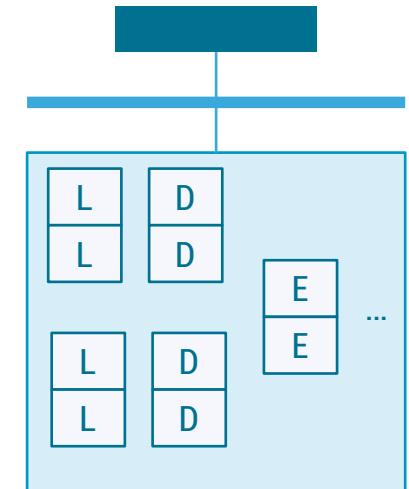


+ exécution dans le désordre,
prédition de branchement,
etc.

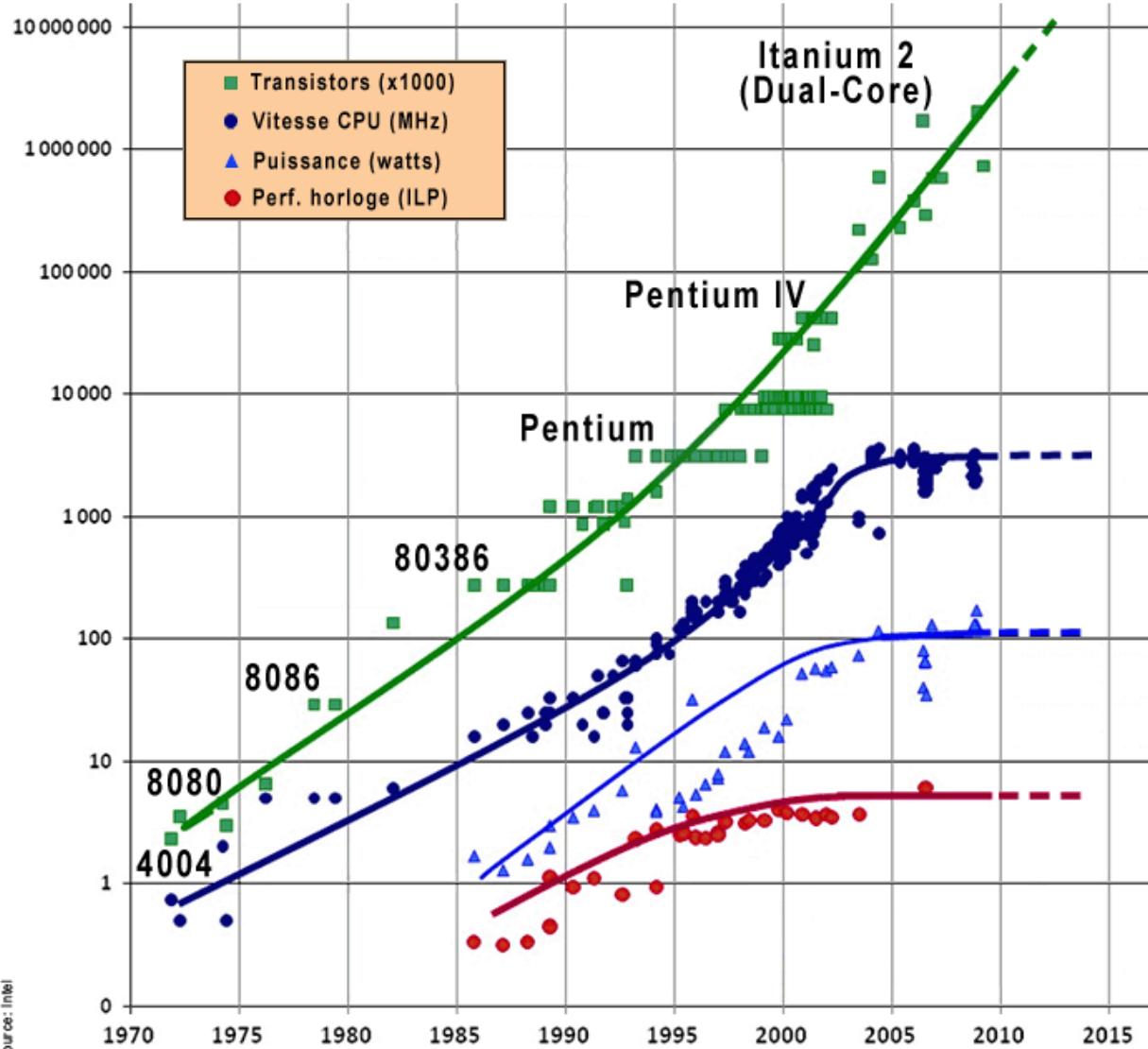
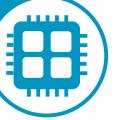
Evolution des processeurs



+ exécution dans le désordre,
prédiction de branchement,
etc.



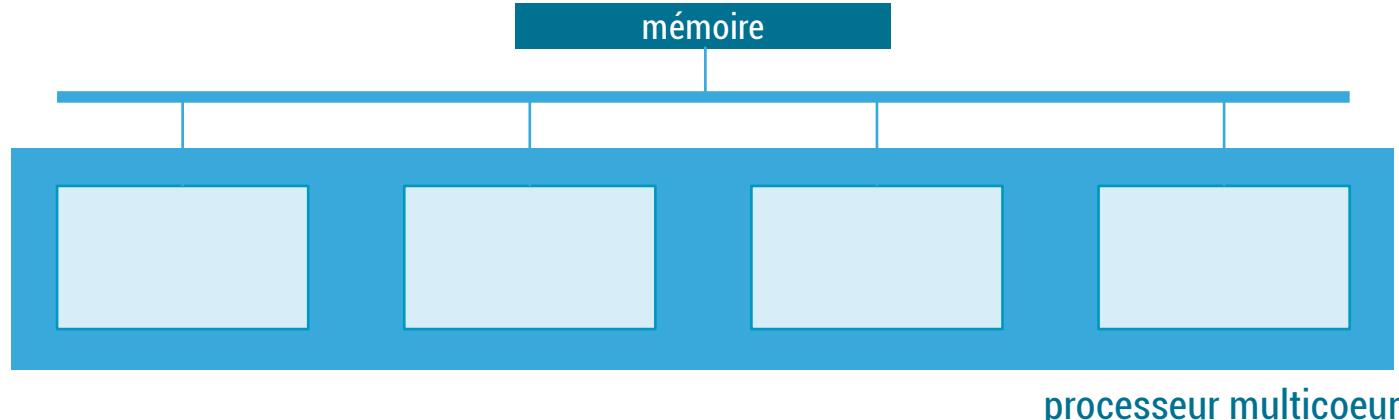
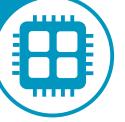
Evolution des processeurs



- **Loi de Moore** : le nombre de transistors augmente exponentiellement
- 3 "murs" :
 - **mémoire** : l'écart entre les performances de la mémoire et du processeur augmente
 - **parallélisme d'instructions** : il n'est pas infini, à cause des dépendances de données
 - **puissance dissipée** : elle augmente avec la fréquence (entre autres)

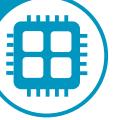


Evolution des processeurs



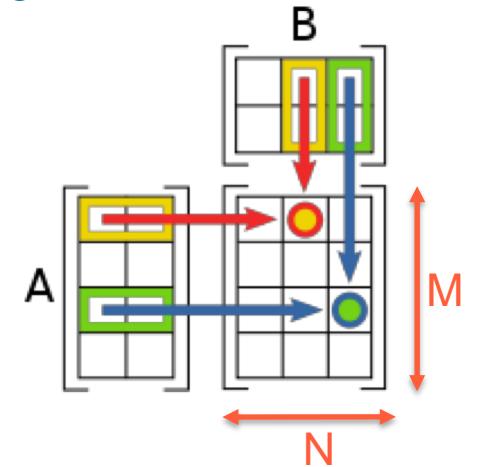
- Plusieurs cœurs (processeurs) indépendants qui utilisent (partagent) la même mémoire
- Capables d'exécuter **simultanément** plusieurs processus indépendants ou plusieurs threads d'un même processus
- Si on veut améliorer les performances d'un programme, il faut le **paralléliser** = le décomposer en plusieurs threads qui vont s'exécuter en parallèle

La programmation parallèle

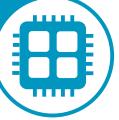


Un moyen de rendre l'exécution d'un calcul plus rapide en créant plusieurs threads qui :

- vont s'exécuter sur des coeurs différents
 - dans ce qui suit, on va considérer un seul thread par cœur (\neq programmation concurrente)
- vont tous contribuer au calcul en réalisant une partie
 - exemple 1 : calcul du produit de deux matrices ($M \times N$)
 - on crée T threads
 - chaque thread calcule M/T lignes de la matrice résultat
 - dans ce cas, les calculs attribués aux threads sont indépendants les uns des autres



La programmation parallèle

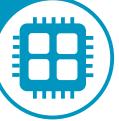


Un moyen de rendre l'exécution d'un calcul plus rapide en créant plusieurs threads qui :

- vont s'exécuter sur des coeurs différents
 - dans ce qui suit, on va considérer un seul thread par cœur
(≠ programmation concurrente)
- vont tous contribuer au calcul en réalisant une partie
 - exemple 2 : somme des éléments d'une matrice ($N \times N$)
 - le résultat est un scalaire
 - plusieurs solutions possibles, par exemple :
 - on crée T threads
 - chaque thread calcule la somme des éléments de N/T lignes
 - un des threads calcule la somme des sommes partielles

communication
synchronisation

Interfaces de programmation parallèle



Pour une mémoire partagée

(espace d'adresses unique pour tous les coeurs)

- threads POSIX OpenMP (ce dont on va parler dans les prochains CTD et TP)
- tous les threads ont accès aux variables partagées en mémoire
- ils peuvent se synchroniser via la mémoire partagée

Pour une mémoire distribuée

(chaque cœur a son propre espace d'adresses)

- MPI : Message Passing Interface (sera vu en M1)
- les threads communiquent par messages (envoi d'une donnée locale, réception d'une donnée distante, etc.)

OpenMP



API de programmation parallèle

- C, C++, Fortran
- spécification produite par un consortium
 - AMD, ARM, Cray, Fujitsu, HP, IBM, Intel, Micron, NEC, NVIDIA, Oracle Corporation, Red Hat, Texas Instruments + quelques universités



openmp.org



computing.llnl.gov/tutorials/openMP

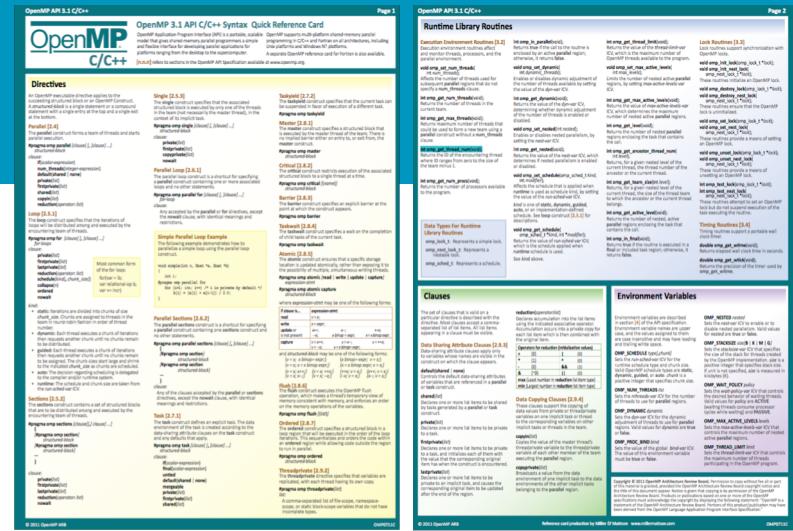
OpenMP

The screenshot shows the main page of the OpenMP website. At the top, there's a navigation bar with links for Home, News, Events, Use, Get, and Learn. The main content area has several sections: "OpenMP News" featuring a recent article about the OpenMP API Specification for Parallel Programming; "OpenMP Events" listing the IWOMP 2016 conference; "Input Register" for vendor contributions; "Search OpenMP.org"; and "Archives" for past news items. On the right side, there's a sidebar with links for "The OpenMP API Specification", "Get", "Use", "Learn", and "Recent News".

OpenMP

3 composantes

- des variables d'environnement
 - exemple : `OMP_NUM_THREADS`
- une bibliothèque de fonctions
 - exemple : `int omp_get_thread_num(void);`
- des directives
 - exemple : `#pragma omp parallel`
 - la plupart des pragmas s'appliquent au bloc structuré qui suit :
`{...}` ou instruction simple



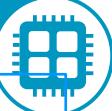
The image shows the OpenMP API 3.1 C/C++ Quick Reference Card, which is a two-page document. The left page covers Directives and Runtime Library Routines, while the right page covers Environment Variables. The card is filled with tables of code snippets and brief descriptions of various OpenMP constructs and functions.

Directives

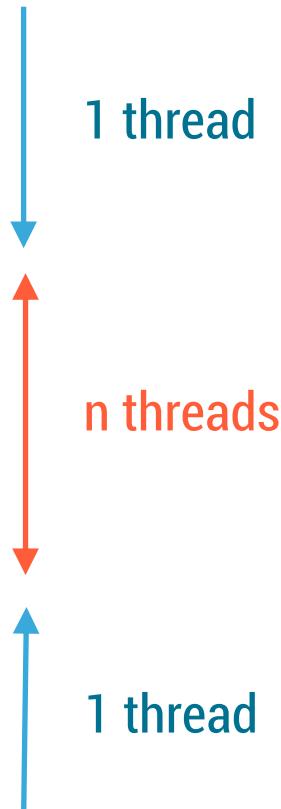
Runtime Library Routines

Environment Variables

Créer des threads en OpenMP



```
int main(){
    // portion de code 1
    ...
    ...
    #pragma omp parallel
    {
        // portion de code 2
        ...
        ...
    }
    // portion de code 3
    ...
    ...
    return(0);
}
```



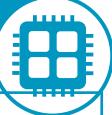
```
pthread_t thd[MAX_NUM_THREADS];

void omp_region1{
    // portion de code 2
    ...
    ...
}

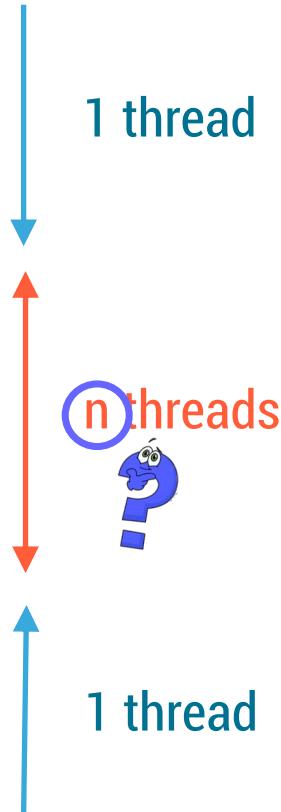
int main(){
    // portion de code 1
    ...
    for (int i=1; i<n; i++)
        pthread_create(&thd[i],NULL,
                      &omp_region1, NULL);
    omp_region1();
    for (int i=1; i<n; i++)
        pthread_join(&thd[i],NULL);
    // portion de code 3
    ...
    return(0);
}
```

TRADUCTION EN PTHREADS

Créer des threads en OpenMP



```
int main(){
    // portion de code 1
    ...
    ...
    #pragma omp parallel
    {
        // portion de code 2
        ...
        ...
    }
    // portion de code 3
    ...
    ...
    return(0);
}
```



n est défini par (dans l'ordre) :

1) une clause :

```
#pragma omp parallel num_threads(8)
{
    ...
}
```

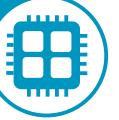
2) l'appel préliminaire à une primitive :

```
omp_set_num_threads(4);
...
#pragma omp parallel
{
    ...
}
```

3) la variable d'environnement
OMP_NUM_THREADS

4) le nombre de coeurs présents

Créer des threads en OpenMP



Exemple

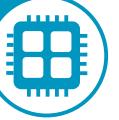
```
#include <omp.h>
#include <stdio.h>

int main(){
    #pragma omp parallel num_threads(4)
    {
        int n = omp_get_thread_num();
        printf("Je suis le thread %d\n",n);
    }
    return(0);
}
```

test1.c

```
> gcc -fopenmp -o test1 test1.c
>
> ./test1
Je suis le thread 0
Je suis le thread 1
Je suis le thread 3
Je suis le thread 2
>
```

Distribuer les calculs entre threads

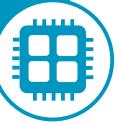


Peut-être contrôlé entièrement par le programmeur ...

- en dirigeant l'exécution en fonction du numéro de thread
 - exemple 1 :

```
int main(){
    #pragma omp parallel num_threads(2)
    {
        int n = omp_get_thread_num();
        if (n==0)
            ... // calcul A
        if (n==1)
            ... // calcul B
    }
    return(0);
}
```

Distribuer les calculs entre threads



Peut-être contrôlé entièrement par le programmeur ...

- en dirigeant l'exécution en fonction du numéro de thread
 - exemple 2 :

```
int main(){
    int tab[4][1024];
    int sum[4];
    init(tab);
    #pragma omp parallel num_threads(4)
    {
        int n = omp_get_thread_num();
        sum[n] = 0;
        for (int i=0; i<1024; i++)
            sum[n] += tab[n][i];
    }
    return(0);
}
```

Distribuer les calculs entre threads



Peut-être contrôlé entièrement par le programmeur ...

- en dirigeant l'exécution en fonction du numéro de thread

... ou géré par le compilateur OpenMP

- en utilisant des pragmas spécifiques
 - `#pragma omp for`
distribue les itérations d'une boucle entre les différents threads actifs
 - `#pragma omp single`
indique qu'un seul thread doit réaliser le calcul (même si plusieurs threads sont actifs)

Distribuer les calculs entre threads



La directive FOR

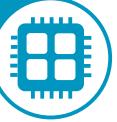
- exemple :

```
#define SIZE 1024
int tab[SIZE];

int main(){
    #pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i<SIZE; i++)
            tab[i] = i;
    }
    return(0);
}
```

- répartit les itérations de la boucle entre les threads actifs
 - il faut que les itérations soient indépendantes les unes des autres !
- tous les threads s'attendent à la fin de la boucle (barrière de synchronisation implicite)

Distribuer les itérations d'une boucle



```
#define SIZE 1024
int tab[SIZE];

int main(){
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for (int i=0; i<SIZE; i++)
        tab[i] = i;
}
return(0);
}
```

Clause schedule :

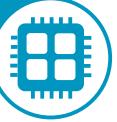
- allocation statique :
 - `schedule(static)` :

thread	0	1	2	3
itérations	0-255	256-511	512-767	768-1023

- `schedule(static,8)` :

thread	0	1	2	3
itérations	0-7 32-39	8-15 40-47	16-23	24-31

Distribuer les itérations d'une boucle

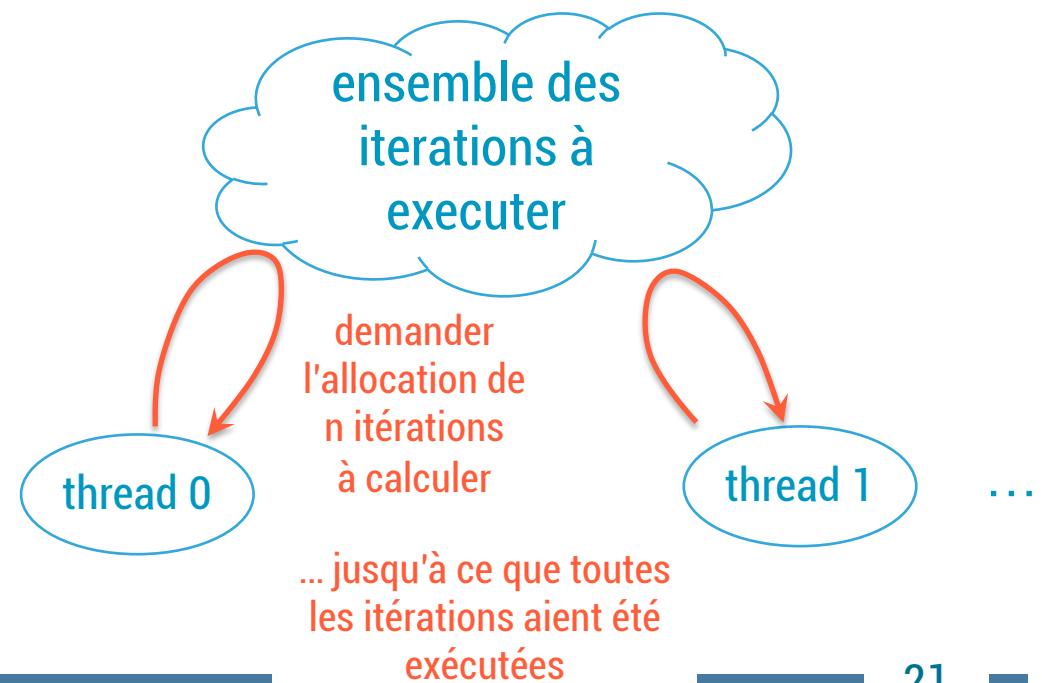


```
#define SIZE 1024
int tab[SIZE];

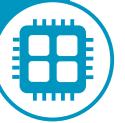
int main(){
#pragma omp parallel num_threads(4)
{
    #pragma omp for
    for (int i=0; i<SIZE; i++)
        tab[i] = i;
}
return(0);
}
```

Clause schedule :

- allocation statique
- allocation dynamique
 - `schedule(dynamic,n)`



Distribuer les calculs entre threads



La directive SINGLE

- exemple :

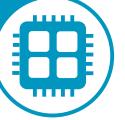
```
#include <omp.h>
#include <stdio.h>

int main(){
    #pragma omp parallel num_threads(4)
    {
        int n = omp_get_thread_num();
        #pragma omp single
        printf("Je suis le thread %d\n",n);
    }
    return(0);
}
```

test2.c

```
> gcc -fopenmp -o test2 test2.c
>
> ./test2
Je suis le thread 2
>
```

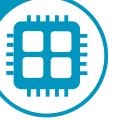
Partage des données



Donnée partagée ou privée

- **partagée :**
 - une seule instance en mémoire, visible par tous les threads
- **privée :**
 - une copie de la donnée pour chaque thread, qu'il est le seul à « voir » (placée dans sa pile)
 - chaque thread peut avoir une valeur différente pour sa donnée privée

Partage des données

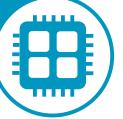


Règles implicites

- les variables déclarées en dehors d'une région parallèle sont en général partagées (**n** et **a**)
- les variable d'itération de boucle sont automatiquement rendues privées par le compilateur (**i**)
 - une copie est créée pour chaque thread
- les variables déclarées à l'intérieur d'une région parallèle sont privées (**b**)

```
int i = 0;  
int n = 10;  
int a = 7;  
  
#pragma omp parallel for  
for (i = 0; i < n; i++){  
    int b = a + i;  
    ...  
}
```

Partage des données

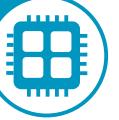


Règles explicites

- clause `shared(<liste de variables>)`
 - les variables de la liste sont partagées
 - peut s'appliquer à une région parallèle ou à une directive de distribution des calculs

```
#pragma omp parallel shared(n,a)
{
    ... // n et a sont partagées par tous les threads
}
```

Partage des données

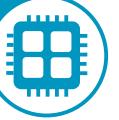


Règles explicites

- clause **shared(<liste de variables>)**
- clause **private(<liste de variables>)**
 - les variables de la liste doivent être rendues privées
 - pas de valeur initiale !

```
int x = 8;
#pragma omp parallel private(x)
{
    ... // chaque thread a sa copie de x, non initialisée
}
... // les copies privées sont détruites, x vaut toujours 8
```

Partage des données

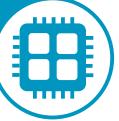


Règles explicites

- clause **shared(<liste de variables>)**
- clause **private(<liste de variables>)**
- clause **default(None)**
 - oblige le programmeur à indiquer explicitement si les variables sont privées ou partagées

```
int a,b;  
#pragma omp parallel default(None) shared(a) private(b)  
{  
    ...  
}
```

Synchroniser la progression des threads



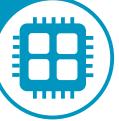
Pourquoi synchroniser la progression des threads ?

- il est parfois nécessaire d'imposer que tous les threads atteignent un certain point du programme avant que tous puissent continuer leur exécution
- notion de **barrière de synchronisation**
- exemple :

```
int number;
#pragma omp parallel
{
    int mynum = omp_get_thread_num();
    if (mynum == 0)
        number = omp_get_num_threads();
    #pragma omp barrier
    printf("Je suis le thread %d parmi %d\n", mynum, number);
}
```

attendre que tous les threads (en particulier le thread 0) soient arrivés là avant de continuer

Synchronisations implicites



Le compilateur ajoute automatiquement une barrière de synchronisation à la suite des directives suivantes :

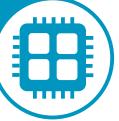
- **#pragma omp parallel**
 - tous les threads sauf le thread maître (thread 0) sont détruits
 - le thread maître doit attendre que tous les autres threads soient terminés avant de continuer l'exécution
- **#pragma omp for**
 - tous les threads doivent attendre que toutes les itérations aient été exécutées
- **#pragma omp single**
 - tous les threads doivent attendre que le bloc de code ait été exécuté

... sauf si la directive est accompagnée d'une clause nowait

- exemple :

```
#pragma omp for nowait  
for (int i=0 ; i<N ; i++)  
    ...
```

Protéger les accès aux données partagées



Exemple d'accès à sécuriser :

```
int V[N];
int num_even = 0; // variable partagée

#pragma omp parallel
{
    #pragma omp for
    for (int i=0 ; i<N ; i++){
        if (V[i] % 2 == 0)
            num_even++;
        // num_even = num_even + 1
    }
}
```

(3) écriture (1) lecture (2) addition

scénario possible :

thread 0

- trouve V[0] pair
- lit num_even (0)
- ajoute 1
- écrit 1 dans num_even

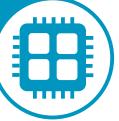
thread 1

- trouve V[128] pair
- lit num_even(0)
- ajoute 1
- écrit 1 dans num_even

résultat : num_even vaut 1 alors que 2 éléments pairs ont été trouvés



Protéger les accès aux données partagées

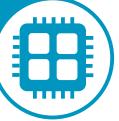


Solution :

- l'opération complète (lecture-modification-écriture) doit être réalisée de manière indivisible
 - quand un thread a commencé la lecture, aucun autre thread ne doit pouvoir réaliser l'opération tant que le 1er n'a pas fini son écriture
- section critique = région de code protégée par un verrou de sorte qu'un seul thread à la fois peut être en train de l'exécuter

```
pthread_mutex_t verrou;  
  
pthread_mutex_lock(&verrou);  
... // instructions de la section critique  
...  
...  
pthread_mutex_unlock(&verrou);
```

Protéger les accès aux données partagées



Retour sur l'exemple :

```
int V[N];
int num_even = 0; // variable partagée

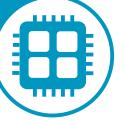
#pragma omp parallel
{
    #pragma omp for
    for (int i=0 ; i<N ; i++){
        if (V[i] % 2 == 0)
            #pragma omp critical
            {
                num_even++;
            }
    }
}
```

- on peut donner un nom à une section critique

```
#pragma omp critical <nom>
```

- toutes les SC de même nom sont protégées par le même verrou
- toutes les SC qui n'ont pas de nom sont protégées par le même verrou

Protéger les accès aux données partagées

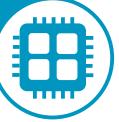


Autre solution :

- utiliser une instruction de bas niveau atomique
 - opération dont l'indivisibilité est garantie par le matériel
 - exemples : compare&swap, fetch&add, test&set
- utilisation de l'instruction compare&swap (cas)
 - arguments :
 - pointeur sur une variable (p)
 - ancienne valeur (av)
 - nouvelle valeur (nv)
 - comportement :
 - si $*p \neq av$ alors
 - renvoyer faux;
 - $*p = nv;$
 - renvoyer vrai;

```
success = false;
ptr = &num_even;
while (!success){
    value = *ptr;
    success = cas(ptr, value, value+1);
}
```

Protéger les accès aux données partagées



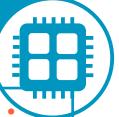
Autre solution :

```
int V[N];
int num_even = 0; // variable partagée

#pragma omp parallel
{
    #pragma omp for
    for (int i=0 ; i<N ; i++){
        if (V[i] % 2 == 0)
            #pragma omp atomic
            num_even++;
    }
}
```

- restrictions :
 - la directive `atomic` ne protège que l'instruction qui suit
 - l'instruction protégée doit être de la forme :
 - `x binop= expr`
 - `x++ , ++x , x-- , --x`
- avec :
 - `x` de type scalaire
 - `expr` = expression de type scalaire qui ne référence pas `x`
 - `binop` = un opérateur binaire (`+, *, -, /, &, ^, |, <<, >>`)

Protéger les accès aux données partagées



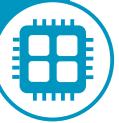
```
#pragma omp atomic  
counter++;
```

```
#pragma omp critical  
counter++;
```

Différences entre les directives `critical` et `atomic` :

- `atomic` ne protège qu'une seule instruction simple
`critical` définit une section critique qui peut contenir plusieurs instructions
- le coût du contrôle d'une section critique (`critical`) est bien plus élevé que celui d'une instruction atomique (`atomic`)
 - mais dans les deux cas, les calculs sont sérialisés
- `critical` protège un bloc d'instructions (qui ne peut être exécuté que par un seul thread à la fois) tandis que `atomic` protège une adresse mémoire

Clause de réduction



Exemple : calcul de la somme des éléments d'un vecteur

```
#define N 2048
int V[N];
int sum= 0;

#pragma omp parallel for num_threads(4)
for (int i=0 ; i<N ; i++)
    #pragma omp atomic
    sum += V[i];
```



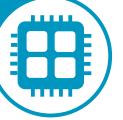
sérialisation des accès à
la variable partagée sum

Solution possible :

- calcul de sommes partielles
- réduction des sommes partielles en une somme globale



Clause de réduction



Exemple : calcul de la somme des éléments d'un vecteur

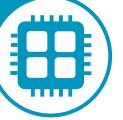
```
#define N 2048
int V[N];
int sum= 0;

#pragma omp parallel for num_threads(4) reduction(+:sum)
for (int i=0 ; i<N ; i++)
    sum += V[i];
```



- crée une copie privée de la variable partagée pour chaque thread (initialisée avec l'élément neutre de l'opérateur, ici 0 pour l'addition)
- les threads font les calculs sur leur copie privée
- à la fin du bloc, chaque thread ajoute sa contribution à la variable partagée

Clauses de réduction



Format

reduction(<opération> : <liste de variables>)



- opérateur binaire : + - * & | ^ && ||
- fonction intrinsèque : max min