

# Programmation Parallèle en Mémoire Partagée

Sophie Robert

Pôle info

## OpenMP

- Vue d'ensemble, modèle de programmation
- Régions parallèles
- Sections critiques, atomiques
- Construction collaboratives : boucle for, réduction
- Synchronisation : barrières, master, single
- Attribut sur le partage des données : shared, private, firstprivate, lastprivate
- **Ordonnancement**

# OpenMP ce n'est pas fini

## OpenMP 2 suite et fin

- Modèle mémoire
- Nouvelle construction : collaboration
- Les verrous

## OpenMP 3

- Tâches

## Quelques éléments OpenMP4

- Gestion de plus bas niveau

# Modèle mémoire

## Mémoire partagée mais ...

- OpenMP propose un modèle à mémoire partagée (SMP - Symmetric shared Memory MultiProcessor).
- Mais les processeurs (matériels) peuvent avoir leur propre mémoire locale rapide (les registres, les caches)
- Si un thread met à jour des données partagées, la nouvelle valeur sera sauvegardée dans un registre, puis stockée dans un cache local
- **La mise à jour n'est pas obligatoirement visible immédiatement par les autres threads !**

# Directive flush

## Cohérence avec la mémoire

La directive `flush` permet de rendre la vue temporaire d'un thread d'une valeur partagée cohérente avec la valeur en mémoire.

## Directive

```
#pragma omp flush ( list )
```

## Interprétation

- Les variables visibles par les threads sont écrites en mémoire après ce point.
- S'il y a des pointeurs dans la liste, seul le pointeur est copié et pas la valeur pointée.

## Construction `collaboration`

La directive `sections` donne un bloc structuré à chaque thread.

- Elle permet de découper un travail constitué de parties indépendantes
- Chaque partie sera réalisée par un thread qui peut éventuellement créer une région parallèle

# Sections principe général

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            x_calculation();
        }
        #pragma omp section
        {
            y_calculation();
        }
        #pragma omp section
        {
            z_calculation();
        }
    }
}
```



## barrière / `nowait`

- Par défaut il y a une barrière à la fin de la construction `omp sections`.
- La clause `nowait` permet de retirer la barrière implicite.



# Le pattern producteur-consommateur

```
#define N 100000
int main(){
    double *A, sum;
    int flag = 0;
    A = new double[N];

    fill_rand(N, A); // Producteur calcule les valeurs d'un
                    // tableau

    sum = Sum_array(N, A); // Consommateur somme les
                          // valeurs du tableau

    return 0;
}
```

# Le pattern producteur-consommateur

```
int main(){
double* A, sum; int numthreads, flag = 0;
A = new double[N];
#pragma omp parallel sections
{
    #pragma omp section // Partie producteur
    {
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush (flag)
    }
}
```

# Le pattern producteur-consommateur

```
#pragma omp section // Partie consommateur
{
    #pragma omp flush (flag)
    while (flag != 1)
        #pragma omp flush (flag)
    #pragma omp flush
    sum = Sum_array(N, A);
}
```

## Conseils

- la directive de collaboration `sections` est adapté à des algorithmes producteur/consommateur.
- **Toujours préférer une directive de type `omp parallel for` si c'est possible**
- **Réserver les sections à du travail clairement découpé en parties indépendantes**

# Les verrous en OpenMP

## Verrous simples

- `omp_lock_t` : Type d'un verrou simple
- `omp_init_lock()` : Pour initialiser un verrou simple
- `omp_set_lock()` : Pour prendre le verrou
- `omp_unset_lock()` : Pour libérer le verrou
- `omp_test_lock()` : Pour tester si le verrou est utilisé ou pas et prendre le verrou s'il est libre
- `omp_destroy_lock()` : Pour détruire le verrou

## fonctionnement

- Le verrou est disponible s'il est **unset**.
- La fonction de verrouillage est bloquante.

# Les verrous en OpenMP

## Barrière mémoire

- Un verrouillage implique une barrière mémoire (*memory fence*) de toutes les variables visibles par le thread (équivalent à un *flush* de toutes les variables).
- Un thread accède toujours à la copie la plus récente d'un verrou. Pas besoin d'utiliser un `flush` sur la variable verrou.

# Exemple de Verrous simples

- 1 Écriture
- 2 Boucle
- 3 Exemple répartition du travail par des `omp_test_lock`

# Encore OpenMP 2 : La clause threadprivate

## threadprivate

Rend une variable privée et la **préserve** pour chaque thread.

## différence avec private

- Clause `private` : la variable globale est masquée
- Clause `threadprivate` : la portée globale de la variable est préservée pour chaque thread.



# Exemple

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

int x = 42;

int main(){

    #pragma omp parallel
    {
        x = omp_get_thread_num();
    }

    #pragma omp parallel
    {
        printf("%d\n", x);
    }
}
```

## Résultat



# Exemple

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

int x = 42;

int main() {

    #pragma omp parallel
    {
        x = omp_get_thread_num();
    }

    #pragma omp parallel
    {
        printf("%d\n", x);
    }
}
```

## Résultat



```
6
6
6
6
6
6
6
6
6
6
```

# Exemple

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

int x = 42;

int main(){
    #pragma omp parallel private(x)
    {
        x = omp_get_thread_num();
    }

    #pragma omp parallel
    {
        printf("%d\n", x);
    }
}
```

## Résultat



# Exemple

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

int x = 42;

int main(){
    #pragma omp parallel private(x)
    {
        x = omp_get_thread_num();
    }

    #pragma omp parallel
    {
        printf("%d\n", x);
    }
}
```

## Résultat

```
42
42
42
42
42
42
42
42
42
```

# Exemple

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

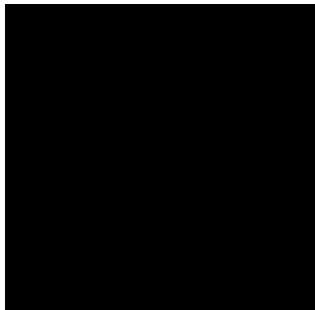
int x = 42;

int main(){

    #pragma omp parallel private(x)
    {
        x = omp_get_thread_num();
    }

    #pragma omp parallel private(x)
    {
        printf("%d\n", x);
    }
}
```

## Résultat



# Exemple

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

int x = 42;

int main(){

    #pragma omp parallel private(x)
    {
        x = omp_get_thread_num();
    }

    #pragma omp parallel private(x)
    {
        printf("%d\n", x);
    }
}
```

## Résultat



```
0
0
0
0
0
0
0
0
```

# Exemple

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

int x = 42;
#pragma omp threadprivate(x)

int main(){

#pragma omp parallel
{
    x = omp_get_thread_num();
}
#pragma omp parallel
{
    printf("%d\n", x);
}
}
```

## Résultat



# Exemple

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

int x = 42;
#pragma omp threadprivate(x)

int main(){

#pragma omp parallel
    {
        x = omp_get_thread_num();
    }
#pragma omp parallel
    {
        printf("%d\n", x);
    }
}
```

## Résultat



```
4
7
0
1
3
5
6
2
```



# Quelques limites d'OpenMP 2

## Structure de données liste

```
class noeud {
private:
    int val;
    noeud* suivant;
public:
    noeud(int _val);
    noeud* next();
    void traiter();
};
class liste {
private:
    noeud* tete;
};
```

# Quelques limites d'OpenMP 2

## Parcours de liste

```
int main(int argc, char* argv[]) {
    liste *l = new liste(); .....
    noeud *n;
    for (n = l->get_tete(); n != NULL; n = n->next())
        n->traiter();
    return 0;
}
```

# Quelques limites d'OpenMP 2

Première solution 

```
#pragma omp parallel
{
    noeud *n;
    for (n=l->get_tete(); n!=NULL; n=n->next()) {
        #pragma omp single nowait
        n->traiter();
    }
}
```

# Quelques limites d'OpenMP 2

## Limites

- Peu naturel
- Mauvaises performances
- Pas composable

# Quelques limites d'OpenMP 2

## Structure de données de type arbre

```
class noeud {
private:
    int val;
    noeud* gauche;
    noeud* droit;
public:
    noeud(int _val);
    noeud* a_gauche();
    noeud* a_droite();
    void traitement();
};
class arbre {
private:
    arbre* racine;
};
```

# Quelques limites d'OpenMP 2

## Parcours séquentiel

```
void noeud::traitement() {  
    if (this->gauche != NULL) gauche->traitement();  
    if (this->droit != NULL) droit->traitement();  
    if (val % 2 == 0)  
        val++;  
    else  
        val += 2;  
}
```

# Quelques limites d'OpenMP 2

## Parcours parallèle



```
void noeud::traitement() {
#pragma omp parallel sections
{
#pragma omp section
    if (this->gauche != NULL) gauche->traitement();
#pragma omp section
    if (this->droit != NULL) droit->traitement();
}
    if (val % 2 == 0)
        val++;
    else
        val += 2;
}
```

# Quelques limites d'OpenMP 2

## Limites

- Trop de régions parallèles : surcoûts, synchronisations supplémentaires
- Pas toujours bien supporté par l'implémentation



# OpenMP 3.0 - OpenMP Tasks

## Une nouvelle directive `task`

```
#pragma omp parallel
{
  .....
#pragma omp task [clauses]
{
  .....
}
.....
}
```

# OpenMP 3.0 - OpenMP Tasks

## Comment fonctionne un tâche/task

- Chaque thread qui **rencontre** la directive task crée une tâche/task
- l'exécution par un des threads de l'équipe créé pour la région parallèle est **immédiate** ou **différée**.
- directive hautement **composable** : imbrication possible dans
  - des régions parallèles
  - d'autres tâches
  - des directives `omp for`, `omp sections`, `omp single`

# Caractéristiques générales des tâches

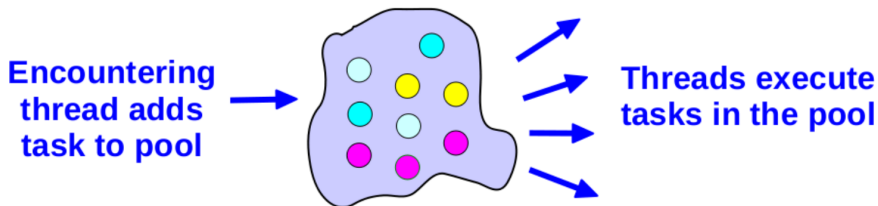
## Une tâche a

- du code à exécuter
- un environnement de données (elle possède ses propres données)
- un thread assigné qui exécute le code et utilise les données

## Deux activités : *packaging* et exécution

- Chaque thread rencontrant la construction `task package` une nouvelle instance de tâche (code + données)
- Un thread de l'équipe exécute la tâche à un moment **ultérieur**

# OpenMP Tasks



***Developer specifies tasks in application***  
***Run-time system executes tasks***

(An Overview of OpenMP 3.0, R. van der Pas, IWOMP2009)

# Tâches dans OpenMP 3.

OpenMP a toujours eu des tâches

Elles avaient juste un autre nom. Le thread rencontrant une construction `parallel package` un ensemble de tâches implicites, une par thread

- 1 Une équipe de thread est créée
- 2 Chaque thread de l'équipe est assigné à une tâche et est lié (tied) à elle
- 3 Une barrière retient le thread maître original jusqu'à ce que toutes les tâches implicites soient terminées.

# Tâches dans OpenMP 3.

## OpenMP a toujours eu des tâches

- ④ OpenMP 3.0 ajoute simplement une manière de créer explicitement une tâche à faire exécuter par une équipe de threads
- ⑤ Toute partie d'un programme OpenMP fait partie d'une tâche!!!

# Construction task

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

## Les clauses

- `if (expression)`
- `untied`
- `shared (list)`
- `private (list)`
- `firstprivate (list)`
- `default( shared | none)`

# Comment utiliser la directive task ?

```
void noeud::traitement() {
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
#pragma omp single nowait
{
#pragma omp task
    if (this->gauche != NULL) gauche->traitement();
#pragma omp task
    if (this->droit != NULL) droit->traitement();
}
}
if (val % 2 == 0)
    val++;
else
    val += 2;
}
```



# La clause `if`

Lorsque l'argument de la clause `if` est faux

- La tâche est exécutée directement par le thread rencontrant la construction `task`
- L'environnement de données est toujours local à la nouvelle tâche ...
- ... et il s'agit toujours d'une tâche différente du point de vue de la synchronisation !

**C'est un outil d'optimisation**

- Lorsque le coût de création d'une tâche est trop élevé en comparaison de son coût d'exécution (test de la taille des données ...)
- Pour contrôler le cache et l'affinité mémoire

# La clause `untied`

## Optimisation

- Par défaut les tâches sont liées
    - Une tâche liée est toujours exécutée par le même thread
  - Conséquence : performance/ordonnancement parfois non optimale
    - Exemple : parcours récursifs -> déséquilibre de charge
  - Clause `untied` et la tâche n'est plus liée à 1 thread donné, mais alors **il faut éviter**
    - les variables `threadprivate`
    - l'utilisation de l'indice du thread dans les calculs de la tâche
- et faire très attention **aux verrous et sections critiques** car des interblocages sont possibles.

# Task switching

## Définition

- Certaines constructions contiennent des points d'ordonnancement de tâches à des positions définies
- Lorsqu'un thread rencontre l'un de ces points, il est autorisé à **suspendre la tâche courante et en exécuter une autre**. C'est du **task switching**.
- Il peut ensuite revenir à la tâche d'origine et la reprendre

# Exemple de *task switching*

les tâches sont **tied**

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

## Interprétation

- Trop de tâches sont générées en un clin d'œil

# Exemple de *task switching*

Les tâches sont **untied**

```
#pragma omp single
{
    for (i=0; i<ONEZILLION; i++)
        #pragma omp task untied
        process(item[i]);
}
```

## Interprétation

- La génération de tâches va être suspendue un moment
- Avec le *task switching*, le thread courant peut :
  - exécuter une tâche déjà générée vidant ainsi le *pool* de tâches
  - exécuter la tâche rencontrée

# Synchronisation pour les tâches

## Quand/où les tâches se terminent ?

- Aux **barrières de threads** explicites ou implicites
  - S'applique à toutes les tâches générées dans la région `parallel` courante jusqu'à la barrière.
- Aux **barrières de tâches**
  - Attend que toutes les tâches définies dans la tâche courante se terminent

```
#pragma omp taskwait
```

- **Attention** : s'applique uniquement aux tâches créées dans la tâche courante, pas à celles créées par ses descendants !

# Quelques exemples - 1. Fibonacci

```
// La parallel region avant l'appel de la fonction
long fibonacci(long n) {
    long x, y;
    if (n<2)
        return n;
    else {
#pragma omp task shared(x) firstprivate(n)
        x=fibonacci(n-1);

#pragma omp task shared(y) firstprivate(n)
        y=fibonacci(n-2);

#pragma omp taskwait
        return x+y;
    }
}
```

# Quelques exemples

Produit de matrices 🙄

- Tasks ou parallel for ?



# Parcours de liste



```
int main(int argc, char* argv[]) {
    liste *l = new liste(); .....
    noeud *n;
    for (n = l->get_tete(); n != NULL; n = n->next())
        n->traiter();
    return 0;
}
```

# Parcours de liste avec des tasks

```
int main(int argc, char* argv[]) {
    liste *l = new liste(); .....
    noeud *n;
#pragma omp parallel
    {
#pragma omp single private(n) nowait
        for (n = l->get_tete(); n != NULL; n = n->next()) {
#pragma omp task untied
            n->traiter();
        }
    }
    return 0;
}
```

# Parcours et traitement pour un arbre

```
#pragma omp parallel
{
#pragma omp single nowait
{
#pragma omp task
    if (this->gauche != NULL) gauche->traitement();
#pragma omp task
    if (this->droit != NULL) droit->traitement();
}
}
if (val % 2 == 0)
    val++;
else
    val += 2;
```

## Encore quelques limitations avec OpenMP 3.

- Il est possible de cibler les architectures parallèles à mémoire partagée (principalement les CPU multi-cœurs).
- On peut définir des régions parallèles, paralléliser l'exécution de boucles, définir et exécuter des tâches parallèles. A partir du programme séquentiel
  - ① on identifie une partie coûteuse (en temps) du programme
  - ② on ajoute une/plusieurs directive(s)/construction(s) OpenMP
  - ③ on optimise (si besoin) avec l'ajout de clauses spécifiques
  - ④ retour en 2

Mais...

On ne profite pas de tout le parallélisme des processeurs modernes (vectorisation)

## Conclusion avec 2 "fonctionnalités" supplémentaires

- Dépendances de tâches
- Programmation vectorielle

# Dépendances de tâches

## Nouvelle clause à la construction `task`

```
depend (dependence-type : list )
```

où `dependence-type` est : `in`, `out` ou `inout`.

## Motivations

- Permettre d'exprimer le parallélisme de tâches d'une manière plus déstructurée.
- Pour potentiellement réduire le nombre de synchronisations coûteuses

# Dépendances de tâches

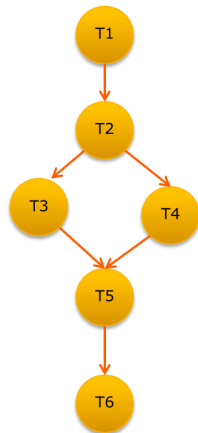
## Comment exprimer les dépendances

Les dépendances sont construites en ordre séquentiel basé sur les relations entre les données

- une tâche avec une dépendance `in:x attend` la fin de toutes les tâches avec une dépendance `out/inout:x`.
- une tâche avec une dépendance `out/inout:x attend` la fin de toutes les tâches avec une dépendance `in/out/inout:x` qui lui précédent aient terminées
- pas de contraintes sur le contenu des tâches
- uniquement valable sur les tâches `sœurs`

# Exemple

```
int a = 10;
#pragma omp task depend(in:a)
  T1
#pragma omp task depend(out:a)
  T2
#pragma omp task depend(in:a)
  T3
#pragma omp task depend(in:a)
  T4
#pragma omp task depend(inout:a)
  T5
#pragma omp task depend(in:a)
  T6
```

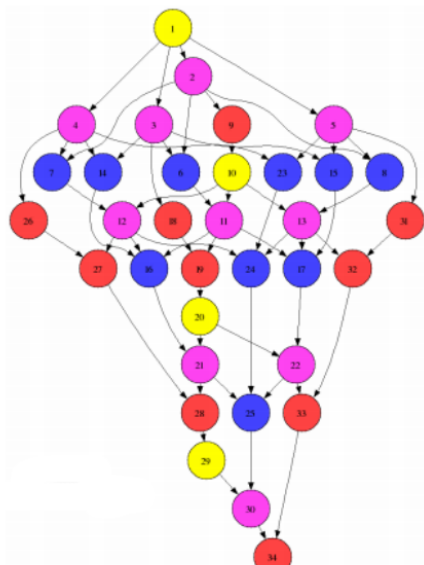




# Exemple plus complexe

```
void blocked_cholesky(int NB, float A[NB][NB]){
    int i, j, k;
    for (k = 0; k < NB; k++){
#pragma omp task depend(inout:A[k][k])
        spotrf(A[k][k]);
        for (i = k+1; i < NT; i++)
#pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
            strsm(A[k][k], A[k][i]);
        //update trailing submatrix
        for (i = k+1; i < NT; i++){
            for (j=k+1; j < i; j++)
#pragma omp task depend(in:A[k][i],A[k][j]) depend(inout:A[j][i])
                sgemm(A[k][i], A[k][j], A[j][i]);
#pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
                ssyrk(A[k][i], A[i][i]);
        }
    }
}
```

# Exemple plus complexe



# Vectorisation

## Calcul scalaire

Un processeur scalaire effectue les opérations séquentiellement, chaque opération portant sur des données scalaires (un scalaire).

## Addition de vecteurs

$$v3[0] = v1[0] + v2[0];$$

$$v3[1] = v1[1] + v2[1];$$

$$v3[2] = v1[2] + v2[2];$$

$$v3[3] = v1[3] + v2[3];$$

# Vectorisation

## Calcul vectoriel

Un processeur vectoriel applique une même instruction simultanément sur plusieurs données (un vecteur de scalaires).

## Addition de vecteurs

$$v3 = v1 + v2;$$

## Modèle SIMD

Single Instruction Multiple Data

## Quelques jeux d'instructions SIMD

x86 : MMX, SSE, AVX

PowerPC : AltiVec

ARM : NEON

# Vectorisation sans OpenMP 4.0

- Les programmeurs doivent compter sur l'auto-vectorisation des compilateurs...
- ...ou utiliser des extensions spécifiques à chaque architecture

## Copie de 4 floats avec SSE

```
#include <stdio.h>
#include <xmmintrin.h> // header pour SSE
int main() {
    float a1[4] __attribute__((aligned(16))) = {1.4, 2.5,
        3.6, 4.8};
    float a2[4] __attribute__((aligned(16)));
    __m128 v1;
    v1 = _mm_load_ps(a1);
    _mm_store_ps(a2, v1);
    printf("%f, %f, %f, %f\n", a2[0], a2[1], a2[2], a2[3]);
    return 0;
}
```

# Construction OpenMP de boucles SIMD

## Une nouvelle directive `simd`

```
#pragma omp simd [clause [[,] clause] ,...]  
suivi de la boucle for
```

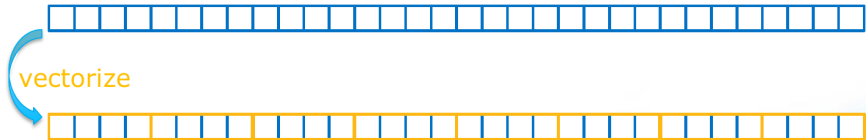
## Vectorise un nid de boucle

- Découpe la boucle en morceaux qui tiennent dans un registre de vecteur SIMD
- Ne parallélise pas le corps de la boucle

# Exemple

## Code

```
void sprod(float *a, float *b, float *c){  
    #pragma omp simd  
    for (int k = 0; k<n; k++)  
        c[k] = a[k] * b[k]  
}
```



# Les clauses de simd

`private(var-list)`

Vecteur de variables non initialisées



`reduction(op:var-list)`

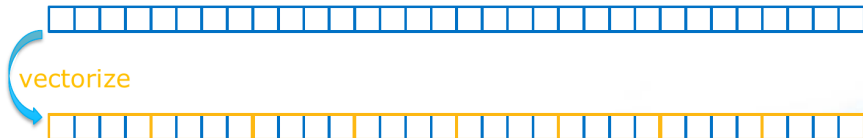
Crée des variables privées pour `var-list` et applique l'opérateur de réduction `op` à la fin de la construction





# Exemple

```
float sprod(float *a, float *b){  
    float sum 0.0f;  
    #pragma omp simd reduction(+:sum)  
    for (int k = 0; k<n; k++)  
        sum+=a[k] * b[k]  
}
```



# Les clauses de simd

## safelen(length)

- Nombre maximum d'itérations qui peuvent s'exécuter en concurrence sans problème de dépendances
- En pratique, taille maximum d'un vecteur

## Exemple

```
for (int i=m; i<n; i++)  
    b[i]=b[i-m]-1.0f;
```

```
#pragma omp simd safelen(m)  
for (int i=m; i<n; i++)  
    b[i]=b[i-m]-1.0f;
```

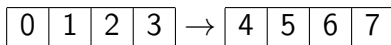
# Les clauses de simd

```
linear(list[:linear-step])
```

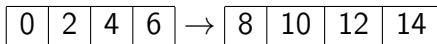
Pour utiliser également la vectorisation pour gérer l'indice de boucle.

## Exemple

- `linear(i:1)`



- `linear(i:2)`



# Construction *worksharing* SIMD

## Parallélise ET vectorise un nid de boucle

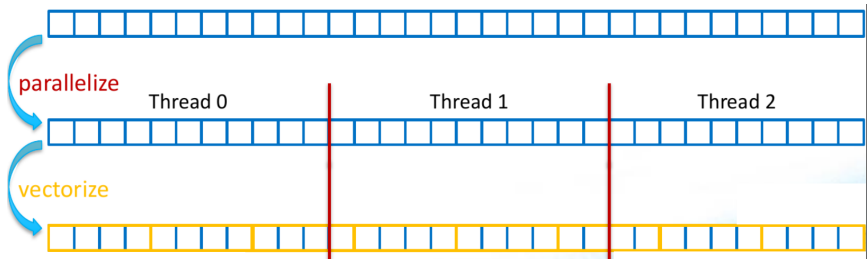
- Distribue l'espace d'itération d'une boucle à travers une équipe de threads
- Subdivise des morceaux de boucle (d'itérations) pour tenir dans un registre vectoriel SIMD

## Syntaxe

```
#pragma omp for simd [ clause [, ] clause ] , ... ]  
boucle for
```

# Exemple

```
void sprod(float *a, float *b, float *c){  
    float sum 0.0f;  
    #pragma omp parallel for simd reduction(+:sum)  
    for (int k = 0; k<n; k++)  
        sum += a[k] * b[k]  
    return sum;  
}
```



# SIMD : vectorisation de fonctions

```
float min(float a, float b){  
    return a<b ? a : b;  
}
```

```
float distsq(float x, float y){  
    return (x-y)*(x-y);  
}
```

```
void example(){  
    #pragma omp parallel for simd  
    for (i = 0; i < n; i++){  
        d[i] = min (distsq(a[i], b[i]), c[i]);  
    }  
}
```

# SIMD : vectorisation de fonctions

## Syntaxe

Déclare une ou plusieurs fonctions pour être compilées afin d'être appelées depuis une boucle parallèle SIMD

```
#pragma omp declare simd  
  declaration ou definition de fonction
```

# SIMD : vectorisation de fonctions

```
#pragma omp declare simd  
float min(float a, float b){  
    return a<b ? a : b;  
}
```

```
#pragma omp declare simd  
float distsq(float x, float y){  
    return (x-y)*(x-y);  
}
```

```
void example(){  
    #pragma omp parallel for simd  
    for (i = 0; i < n; i++){  
        d[i] = min (distsq(a[i], b[i]), c[i]);  
    }  
}
```