

# Une introduction à OpenMP

Franck Cappello  
INRIA – LRI, Université Paris sud

Certains transparents proviennent du cours  
de Marc Snir

# Quelques pointeurs

- Le site officiel du ‘standard’ OpenMP (en Anglais)
  - <http://www.openmp.org>
- Les cours de l’IDRIS (en Français)
  - <http://www.idris.fr/data/cours/parallel/openmp/>

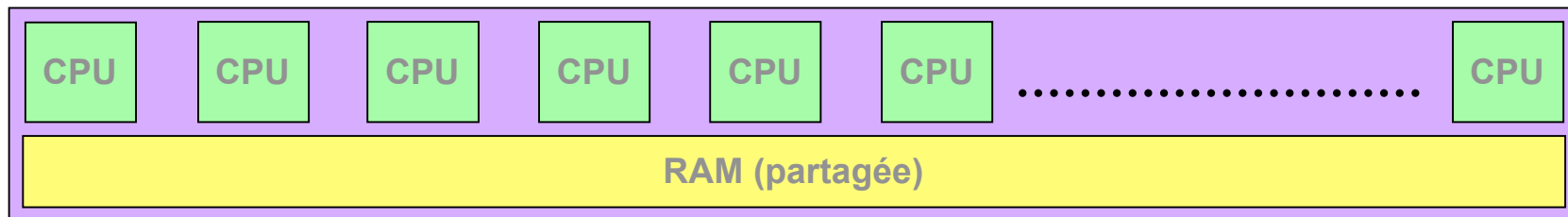
# Sommaire

- Introduction
- L'API OpenMP
  - Les directives
  - Bibliothèque de fonctions
  - Variables d'environnement
- Un petit exemple
- Quelques difficultés
- Conclusion

# Multithreading en mémoire partagée (1)

- Le problème :
  - On dispose de  $n$  processeurs
  - Ces machines sont connectées d'un mécanisme de mémoire partagée entre eux

➔ *Comment utiliser cette machine à  $n$  processeurs reliés par de la mémoire partagée ?*



# Multithreading en mémoire partagée (2)

- Une réponse : le multithreading en mémoire partagée
  - Faire exécuter un processus qui va spawner des threads (flots d'exécution) sur chaque processeur
  - Effectuer des échanges de données de façon transparente via la mémoire partagée
  - Synchroniser explicitement au besoin, par exemple avec des verrous (*locks*)
- Le modèle le plus simple pour l'utilisateur paraît être d'ouvrir/fermer les sections parallèles en fonction des besoins et de la possibilité de paralléliser

# Origine d'OpenMP

Propriétés requises pour un environnement de programmation parallèle :

- portable
- extensible
- efficace
- programmation de haut niveau
- facilite le parallélisme de données
- relativement simple à utiliser (pour les codes existants et les nouveaux)

Environnements existants

- Bibliothèque de Threads bas niveau
  - Win32 API, Posix thread.
- Bibliothèque de passage de messages
  - MPI, PVM
- Langages de haut niveau
  - HPF, Fortran D, approche orienté objet
- Directives de compilateur
  - Un compilateur par constructeur.
  - Initiative X3H5 vers la définition d'un standard AINSI

# OpenMP : Introduction

*OpenMP: une API pour écrire des Applications Multithread (spécification de 1997)*

- Un ensemble de directives de compilation et une bibliothèque de fonctions
- Facilite la création de programmes multi-thread (MT) Fortran, C and C++
- Standardise la pratique de la programmation des machines SMP des 15 dernières années

Défendu par

- les principaux constructeurs
- les vendeurs de logiciels (compilateurs entre autres)
- les vendeurs d'applications

# Mode de programmation

OpenMP est utilisé « principalement » pour paralléliser les boucles :

- Trouver les boucles les plus coûteuses en temps
- Distribuer leurs itérations sur plusieurs threads.

Eclater cette boucle vers  
plusieurs threads

```
void main()
{
    double Res[1000];

    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

Programme Séquentiel



```
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

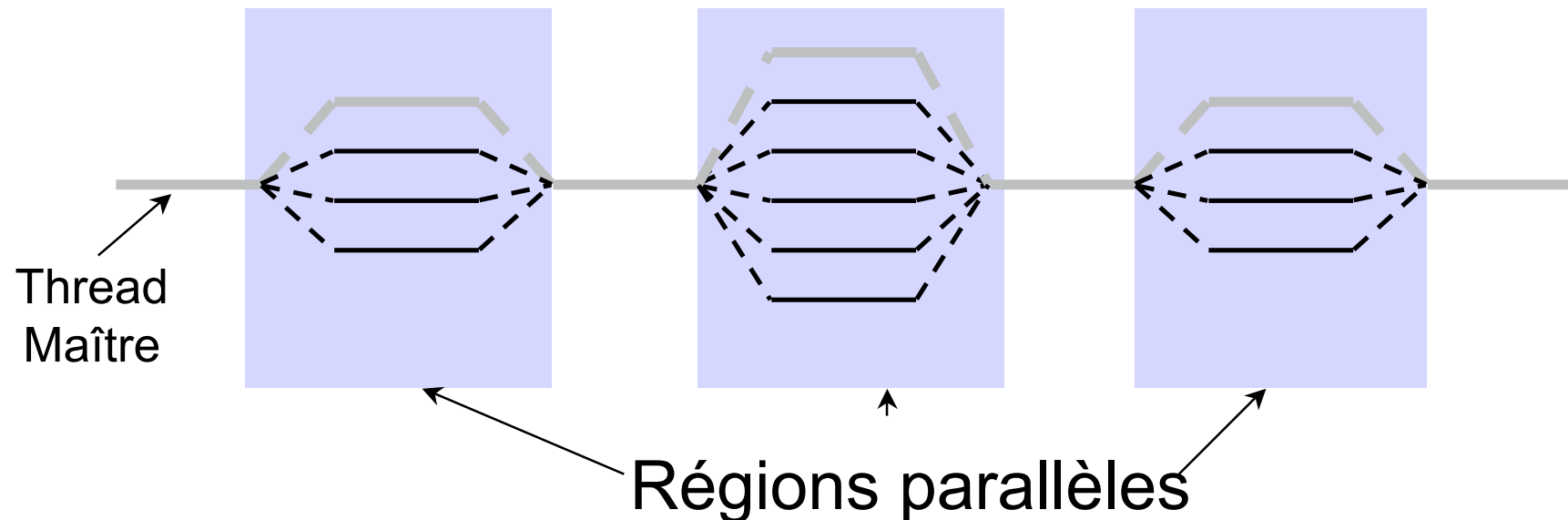
Programme Parallèle



# OpenMP : Modèle d'exécution

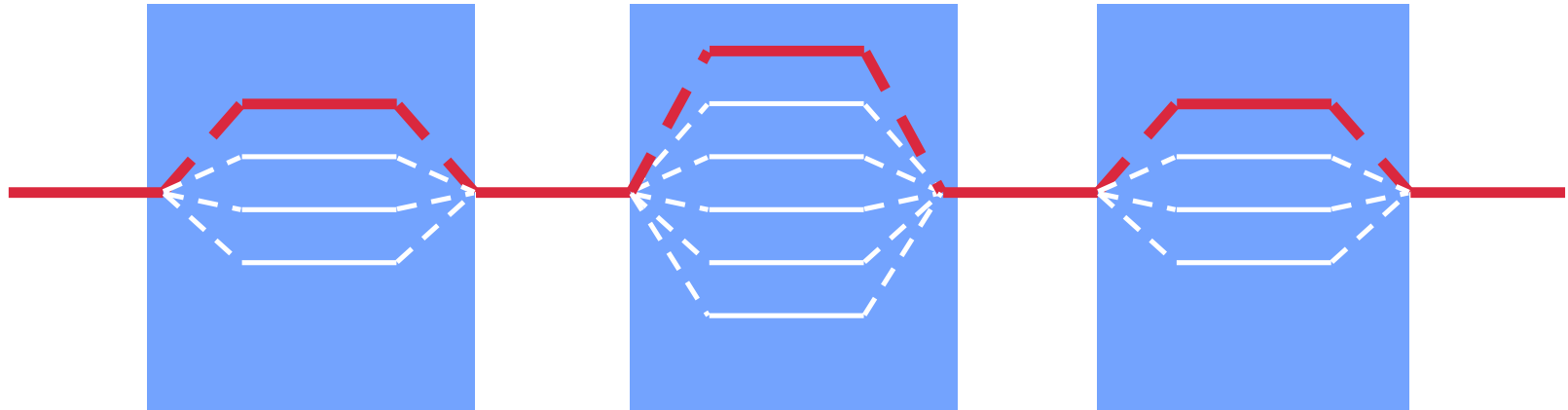
## Parallélisme Fork-Join :

- ◆ Le **thread Maître** lance une **équipe de threads** selon les besoins (région parallèle).
- ◆ Le parallélisme est introduit de façon incrémentale ; le programme séquentiel évolue vers un prog. parallèle

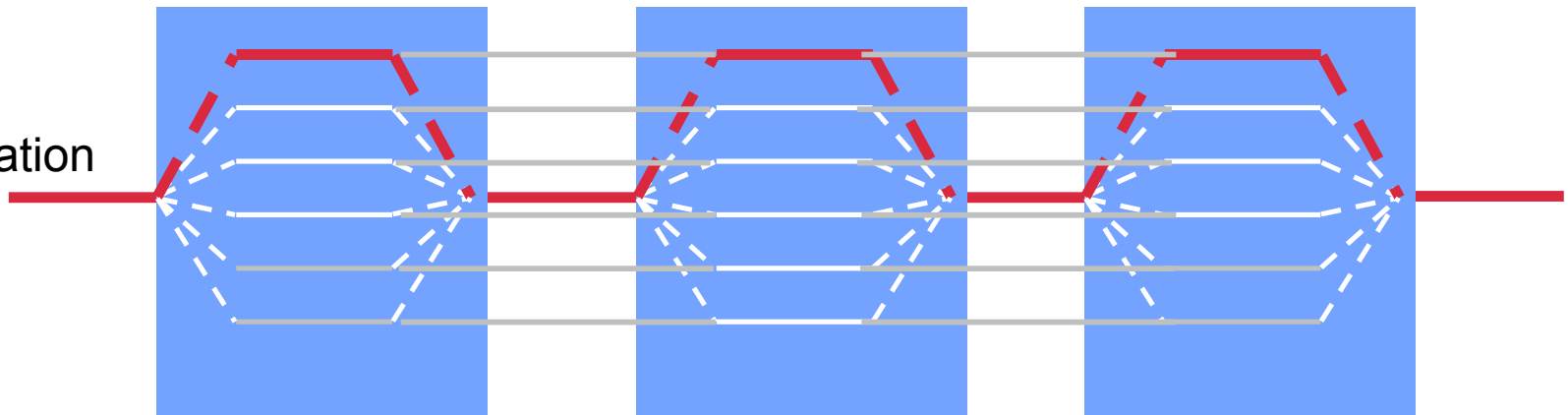


# Modèle abstrait Fork-Join != appels Unix fork-join

Modèle  
abstrait

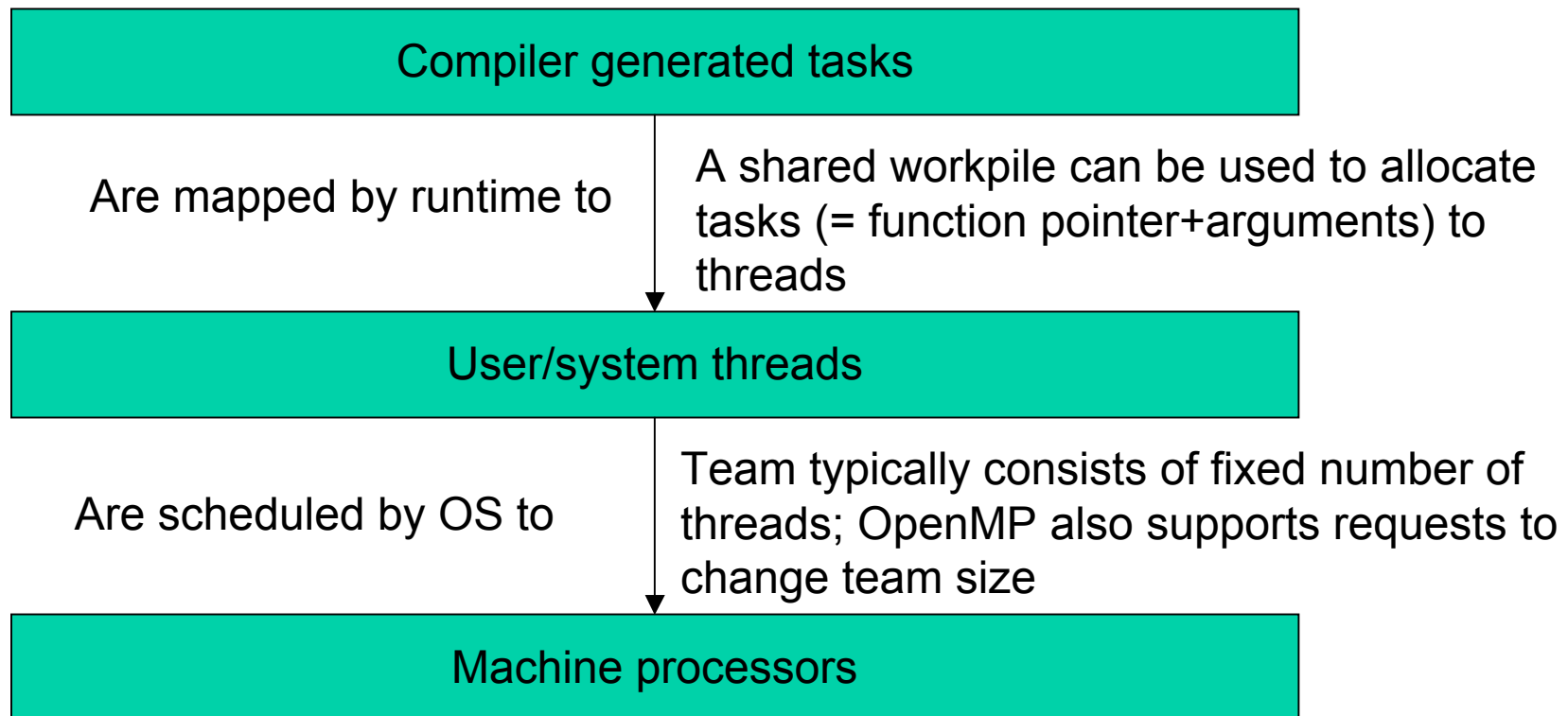


Implementation  
possible



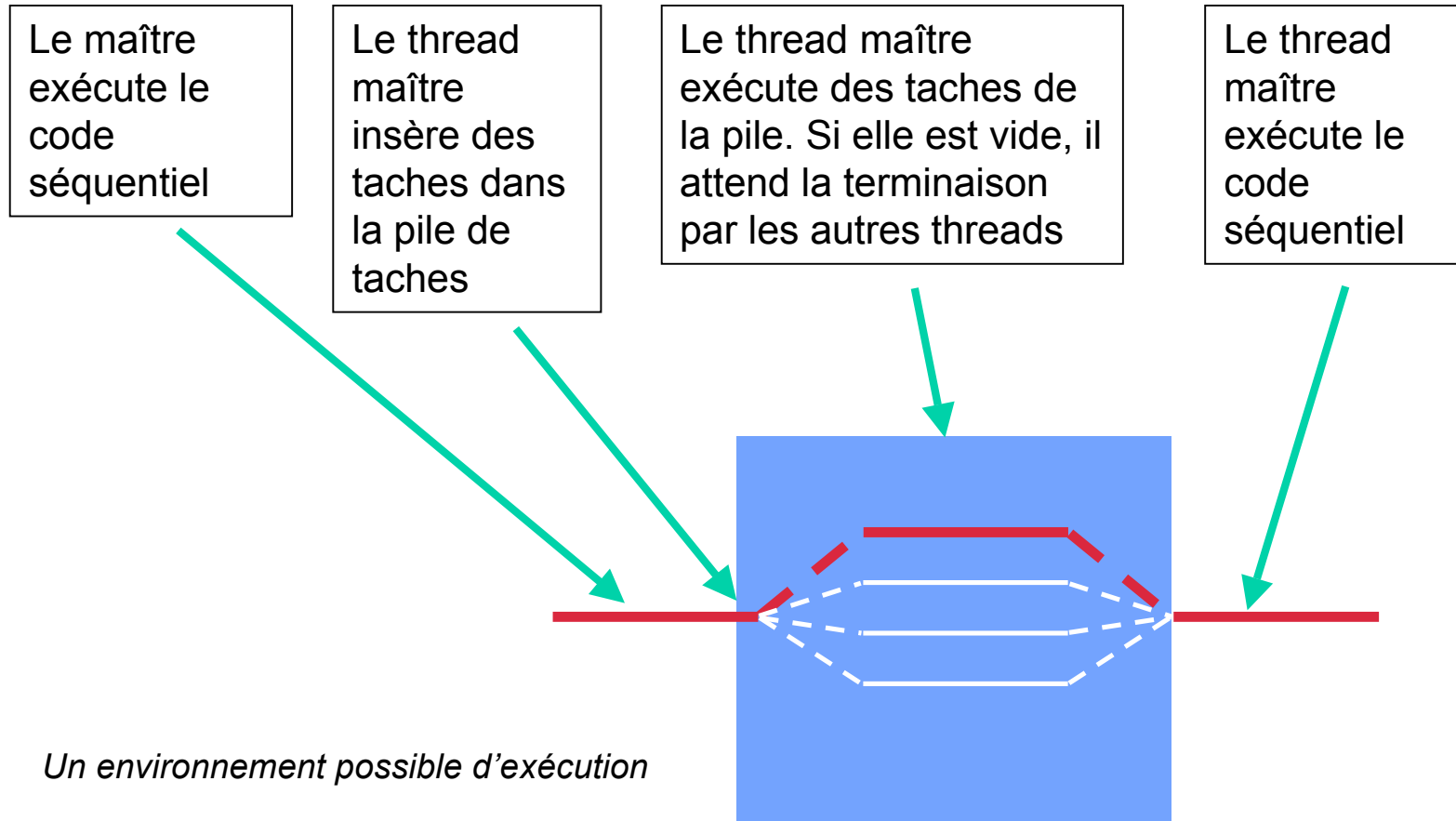
# Two-level scheduling

Compiler typically translates task into a procedure invocation



OS may support gang scheduling and affinity scheduling; one may be able to get a set of (mostly) dedicated processors

# Implémentation possible



Les threads esclaves exécutent la boucle : {prendre tâche dans la pile; exécuter la tâche}

# OpenMP : détails de syntaxe

- La plupart des constructeurs OpenMP sont des directives en commentaires (ou pramas).
  - C et C++ :  
*#pragma omp construct [clause [clause]...]*
  - Fortran :  
*C\$OMP construct [clause [clause]...]*  
*!\$OMP construct [clause [clause]...]*  
*\*\$OMP construct [clause [clause]...]*
- Un programme OpenMP peut être compilé par un compilateur non OpenMP
- Les compilateurs OpenMP offrent généralement la possibilité de compiler un programme sans interpréter les directives OpenMP (*permet la comparaison rapide entre parallélisation automatique et parallélisation par directives*)

# OpenMP : blocks de base

Les constructeurs OpenMP s'appliquent à des blocs structurés.

Un bloc structuré : un bloc de code avec un seul point d'entrée (au début du bloc) et un seul point de sortie (à la fin du bloc).

```
!$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
!$OMP END PARALLEL
      print *,id
```

Un bloc structuré

```
      if(conv(res(id)) goto 30
!$OMP PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
      if(conv(res(id)) goto 20
      go to 10
!$OMP END PARALLEL
      if(not_DONE) goto 30
20    print *, id
```

Un bloc non structuré

# L ' API OpenMP

- Les constructeurs OpenMP :
  - Les régions parallèles
  - La distribution du calcul
  - L'environnement de données
  - La synchronisation
- Bibliothèque de fonctions
- Variables d'environnement

# Les régions parallèles

Les threads sont créés avec la directive “omp parallel”.

```
!$OMP PARALLEL [CLAUSE[[,] CLAUSE] ...]
```

...

```
!$OMP END PARALLEL
```

Par exemple, pour créer région Parallèle à 4 threads :

Chaque thread exécute de manière redondante le code à l'intérieur du bloc structuré

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID = omp_thread_num();  
    fonc(ID,A);  
}
```

Tous les threads attendent ici les autres threads avant de continuer (*barrier*)

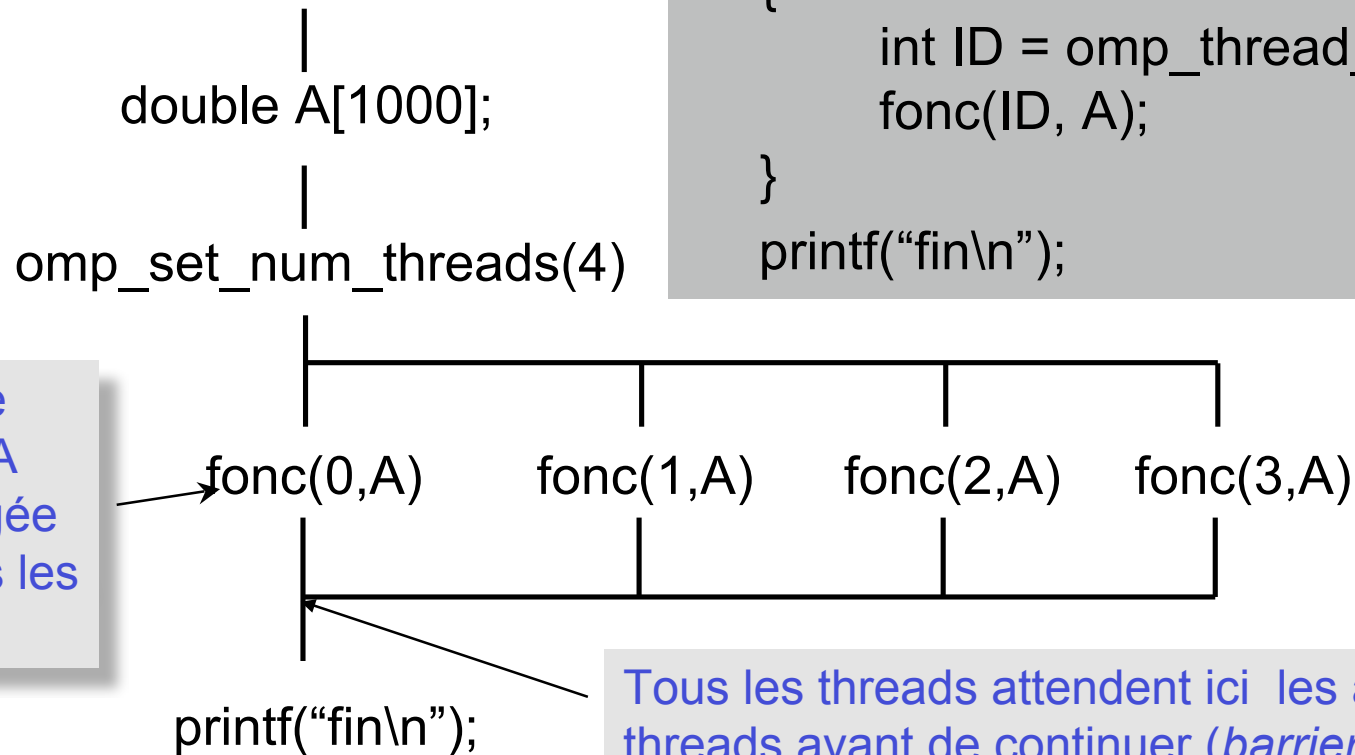
Chaque thread appelle `fonc(ID,A)` avec `ID` différent



# Les régions parallèles

Tous les threads exécutent le même code.

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID = omp_thread_num();  
    fonc(ID, A);  
}  
printf("fin\n");
```



Une seule copie de A est partagée entre tous les threads.

Tous les threads attendent ici les autres threads avant de continuer (*barrier*)

# Distribution du calcul

Les constructeurs “for” pour C et C++ et “do” pour Fortran distribuent les itérations de la boucle sur les différents threads de l’équipe

```
!$OMP DO [CLAUSE [ [, ] CLAUSE] ... ]  
    boucle DO  
!$OMP ENDO
```

Exemple de code  
séquentiel

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i] ; }
```

Constructeur  
OpenMP pour  
région parallèle  
et distribution  
du calcul

```
#pragma omp parallel  
#pragma omp for  
    for(i=0;i<N;i++) { a[i] = a[i] + b[i] ; }
```

Par défaut, il y a une barrière à la fin de la boucle. La clause “nowait” élimine la barrière

# Distribution des itérations sur les threads

! \$OMP DO

SCHEDULE ( type [ , CHUNK ] )

- La clause schedule définit comment les itérations de la boucle sont placées sur les threads
  - **schedule(static [,chunk])**
    - distribue des blocs d 'itérations de taille “chunk” sur chaque thread.
  - **schedule(dynamic[,chunk])**
    - Chaque thread collecte dynamiquement “chunk” itérations dans une queue jusqu'à ce que toutes les itérations soient exécutées.
  - **schedule(guided[,chunk])**
    - Les threads collectent dynamiquement des blocs d 'itérations. La taille des blocs diminue selon une fonction exponentielle jusqu 'à la taille chunk.
  - **schedule(runtime)**
    - Le mode de distribution et la taille des blocs sont définis par la variable d 'environnement OMP\_SCHEDULE.

# Distribution du calcul

Le constructeur **Sections** attribue à chaque thread un bloc structuré différent. Programmation **MPMD**.

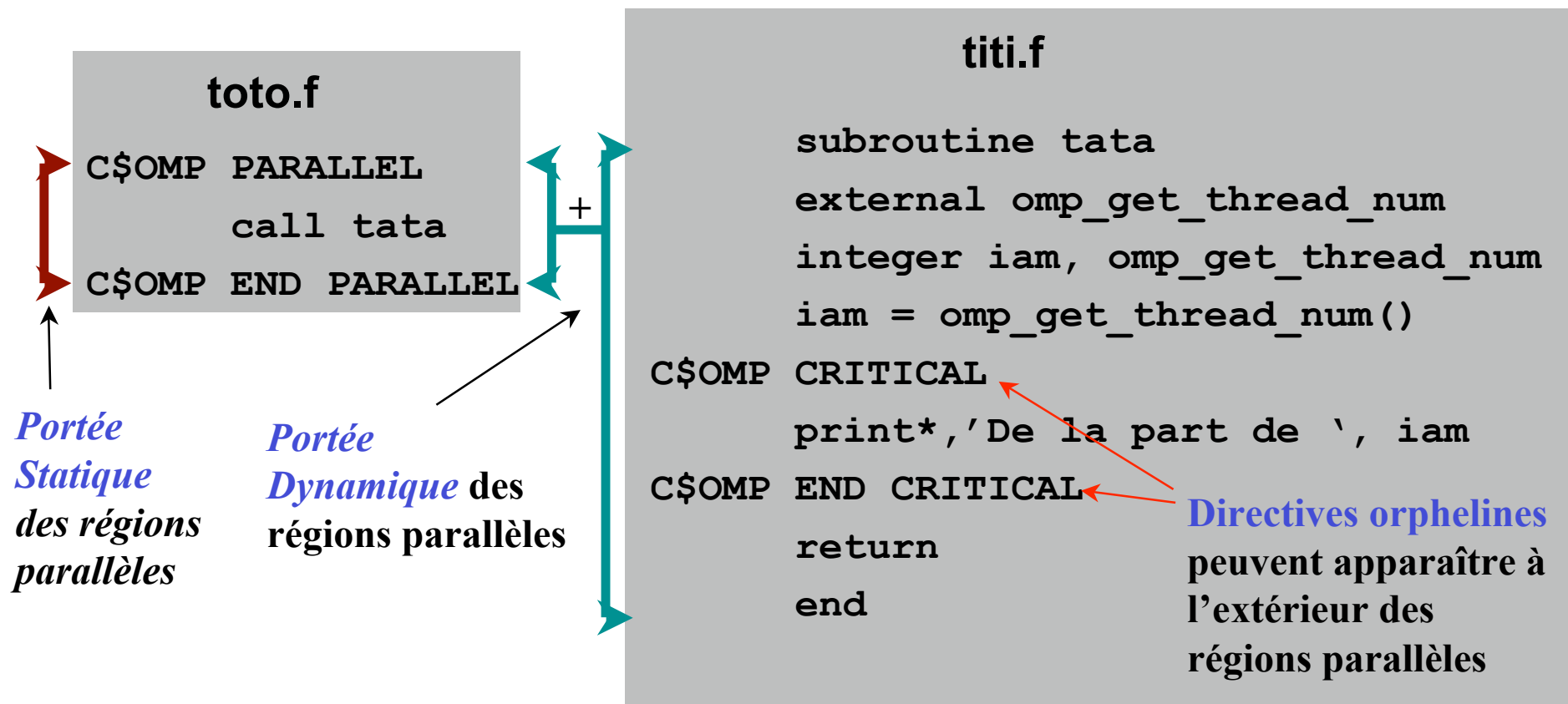
```
!$OMP SECTION [CLAUSE [, ] CLAUSE] ...]
```

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

Par défaut, il y a une « barrier » à la fin du bloc “section”. La clause “nowait” élimine la barrière

# Portée des constructeurs OpenMP

La portée (champ d'application) des constructeurs OpenMP s'étend au delà des frontières de fichiers sources.



# Environnement de données

## Modèle de programmation à mémoire partagée :

La plupart des variables sont partagées par défaut

## Les variables globales sont partagées par les threads

Fortran: blocs COMMON, variables SAVE et MODULE

C: variables de fichier, static

## Tout n'est pas partagé

Les variables de la pile des fonctions appelées à l'intérieur d'une section parallèle sont privées

```
!$OMP PARALLEL [CLAUSE[[,] CLAUSE]...
```

```
!$OMP DO [CLAUSE[[,] CLAUSE]...
```

```
!$OMP SECTION [CLAUSE[[,] CLAUSE]...
```

# Environnement de données

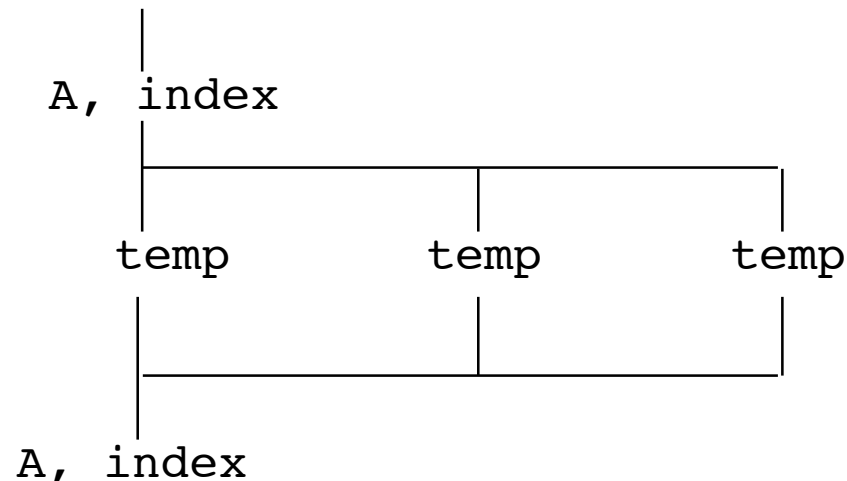
## attributs par défaut

```
program tri
common /input/ A(10)
integer index(10)
call input
!$OMP PARALLEL
  call toto(index)
!$OMP END PARALLEL
print*, index(1)
```

```
subroutine toto
real temp(10)
.....
```

**A, et index sont  
partagés par tous les  
threads.**

**temp est local à  
chaque thread**



# Environnement de données

## Changer l'attribut de stockage

- Les clauses suivantes modifient l'attribut de stockage :
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE
- Une valeur privée à l'intérieur d'une boucle parallèle peut être transmise à une valeur globale à la fin de la boucle :
  - LASTPRIVATE
- l'attribut par défaut peut être chargé :
  - DEFAULT (PRIVATE | SHARED | NONE)

toutes les clauses s'appliquent aux régions parallèles et aux constructeurs de distribution du calcul sauf "shared" qui concerne seulement les régions parallèles



# Réduction

```
!$OMP PARALLEL REDUCTION({opér., intr.}:liste)
```

```
!$OMP DO REDUCTION ({opér., intr.}:liste)
```

- La clause “reduction” permet de réaliser une opération de réduction sur une variable:
  - reduction (op : liste)      op : +, -, \*, max, min, or, etc.
- Les variables dans la liste doivent être des scalaires partagés à l’intérieur d’une région parallèle.
- Utilisable dans les directives “parallel” et de distribution de calcul :
  - Une copie locale de chaque variable dans la liste est attribuée à chaque thread et initialisée selon l’opération à réaliser (0 pour “+”)
  - Les réductions intermédiaires sur chaque thread sont “visibles” en local
  - Les copies locales sont réduites dans une seule variable globale à la fin de la section.

# Exemple de réduction

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
    int i;
    double ZZ, func(), res=0.0;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel for reduction(+:res) private(ZZ)
    for (i=0; i< 1000; i++){
        ZZ = func(i);
        res = res + ZZ;
    }
}
```

# Trace d'une matrice (1)

- ❑ Calcul de la trace d'une matrice  $A_n$
- ❑ Rappel : la trace d'une matrice est la somme des éléments de sa diagonale (matrice nécessairement carrée)
- ❑ Mathématiquement, on sait que :  $Trace(A) = \left( \sum_{k=1}^n A_{k,k} \right)$
- ❑ Immédiatement, on voit facilement que le problème peut être parallélisé en calculant la somme des éléments diagonaux sur plusieurs processeurs puis en utilisant une réduction pour calculer la trace globale

# Trace d'une matrice (2)

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

void main(int argc, char ** argv) {
    int me, np, root=0;
    int N; /* On suppose que N = m*np */
    double A[N][N];
    double traceA = 0;

    /* Initialisation de A */
    /* ... */
    #pragma omp parallel default(shared)
    #pragma omp for reduction(+:trace) {
        for (i=0; i<N; i++) {
            traceA += A[i][i]
        }
    }
}
```

```
        printf("La trace de A est : %f \n",
               traceA);
    }
```

# Un exemple

```
subroutine jacobi
(n,m,dx,dy,alpha,omega,u,f,tol,maxit)
...
!$omp parallel
!$omp do
    do j=1,m
        do i=1,n
            uold(i,j) = u(i,j)
        enddo
    enddo
!$omp enddo
* Compute stencil, residual, & update
!$omp do private(resid) reduction(+:error)
    do j = 2,m-1
        do i = 2,n-1
            resid = (ax*(uold(i-1,j) + uold(i+1,j))
&                  + ay*(uold(i,j-1) + uold(i,j+1))
&                  + b * uold(i,j) - f(i,j))/b
* Update solution
            u(i,j) = uold(i,j) - omega * resid
* Accumulate residual error
            error = error + resid*resid
        end do
    enddo
!$omp enddo
!$omp end parallel
```

# Synchronisation

OpenMP possède les constructeurs suivants pour les opérations de synchronisation

- atomic
- critical section
- barrier
- flush
- ordered

- single
- master



Pas vraiment des opérations de synchronisation

# Synchronisation

- **Atomic** est un cas spécial de section critique qui peut être utilisé pour certaines opérations simples.
- Elle s'applique seulement dans le cadre d'une mise à jour d'une case mémoire

```
!$OMP PARALLEL PRIVATE(B)
      B = DOIT(I)
!$OMP ATOMIC
      X = X + B
!$OMP END PARALLEL
```

# Synchronisation

Un seul thread à la fois peut entrer dans une section critique (**critical**)

```
!$OMP PARALLEL DO PRIVATE(B)
!$OMP& SHARED(RES)
    DO 100 I=1,NITERS
        B = DOIT(I)
!$OMP CRITICAL
            CALL COMBINE (B, RES)
!$OMP END CRITICAL
    100    CONTINUE
!$OMP END PARALLEL DO
```



# Synchronisation

**BARRIER** tous les thread attendent que les autres threads arrivent au même point d'exécution avant de continuer

```
#pragma omp parallel shared (A, B, C) private(id)
{
```

```
    id=omp_get_thread_num();
```

```
    A[id] = big_calc1(id);
```

```
#pragma omp barrier
```

```
#pragma omp for
```

```
    for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
```


```
#pragma omp for nowait
```

```
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
```

```
    A[id] = big_calc3(id);
```

```
}
```

Barrière implicite à la fin de la construction omp for



implicit barrier at the end of a parallel region



Pas de barrière implicite nowait



# Synchronisation : flush

- Exemple d'utilisation de **flush** (consistance mémoire)

```
!$OMP PARALLEL SECTIONS
```

```
  A = B + C
```

```
!$OMP SECTION
```

```
  B = A + C
```

```
!$OMP SECTION
```

```
  C = B + A
```

```
!$OMP END PARALLEL SECTIONS
```

```
  ICOUNT = 0
```

```
!$OMP PARALLEL SECTIONS
```

```
  A = B + C
```

```
  ICOUNT = 1
```

```
!$OMP FLUSH ICOUNT
```

```
!$OMP SECTION
```

```
1000 CONTINUE
```

```
!$OMP FLUSH ICOUNT
```

```
  IF(ICOUNT .LT. 1) GO TO 1000
```

```
  B = A + C
```

```
  ICOUNT = 2
```

```
!$OMP FLUSH ICOUNT
```

```
!$OMP SECTION
```

```
2000 CONTINUE
```

```
!$OMP FLUSH ICOUNT
```

```
  IF(ICOUNT .LT. 2) GO TO 2000
```

```
  C = B + A
```

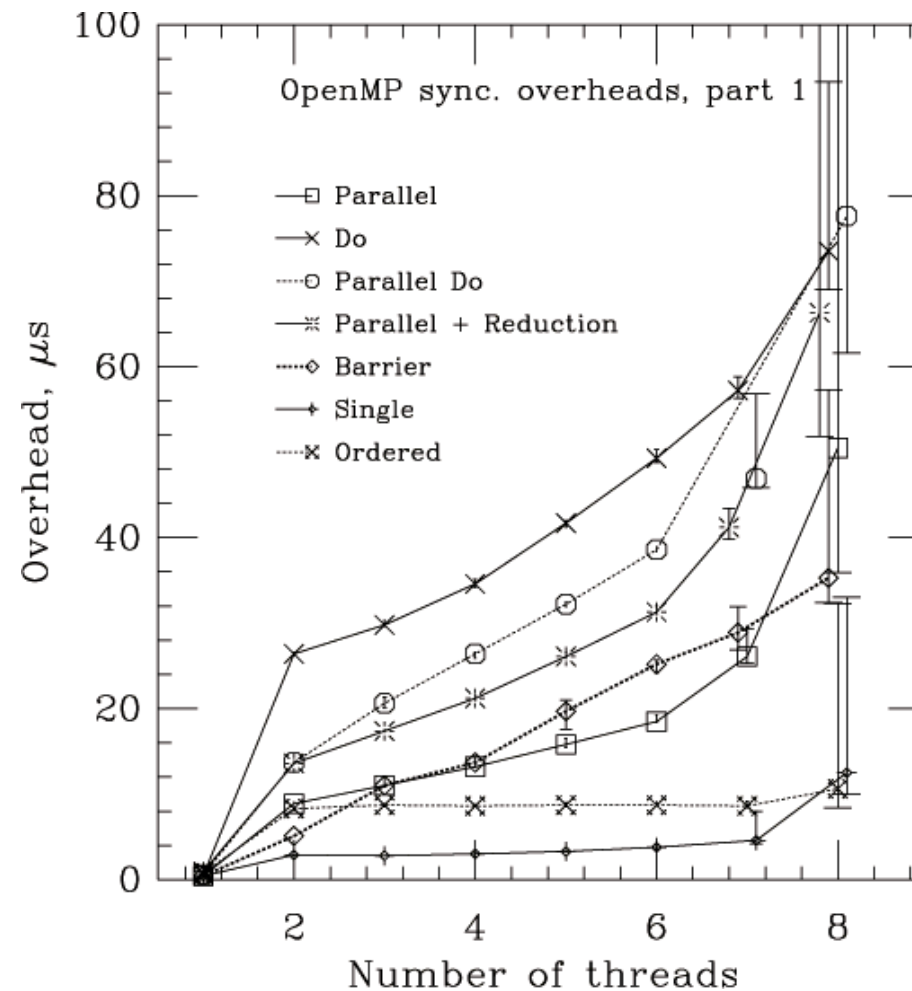
```
!$OMP END PARALLEL SECTIONS
```

# Synchronisation : SINGLE / MASTER

- Ces directives permettent d'assurer que le code n'est exécuté que par un seul thread
- Dans le cas de SINGLE, c'est le premier thread qui atteint la zone considérée qui procède à l'exécution
- Dans le cas de MASTER, c'est toujours le thread maître (de rang 0) qui exécute la tâche

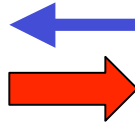
# Coût des synchronisations

mesures sur IBM SP3 NH1(SDSC), compilateur IBM



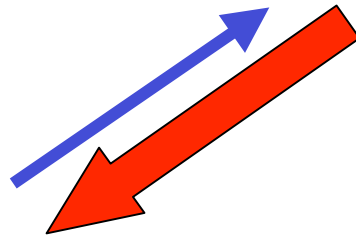
# Balance : taille de régions parallèles VS nombre de synchronisations

Problème  
de performance

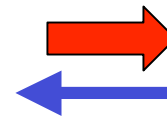


Augmenter la taille de la ou des régions  
parallèles :

- > plus de code « scalable »
  - > moins de « fork & join »
- mais plus de variables partagées



Augmentation du nombre de  
synchronisations dans la région  
parallèle **OU**  
Utilisation de variables  
supplémentaires



- Baisse de performance
- Développement plus long



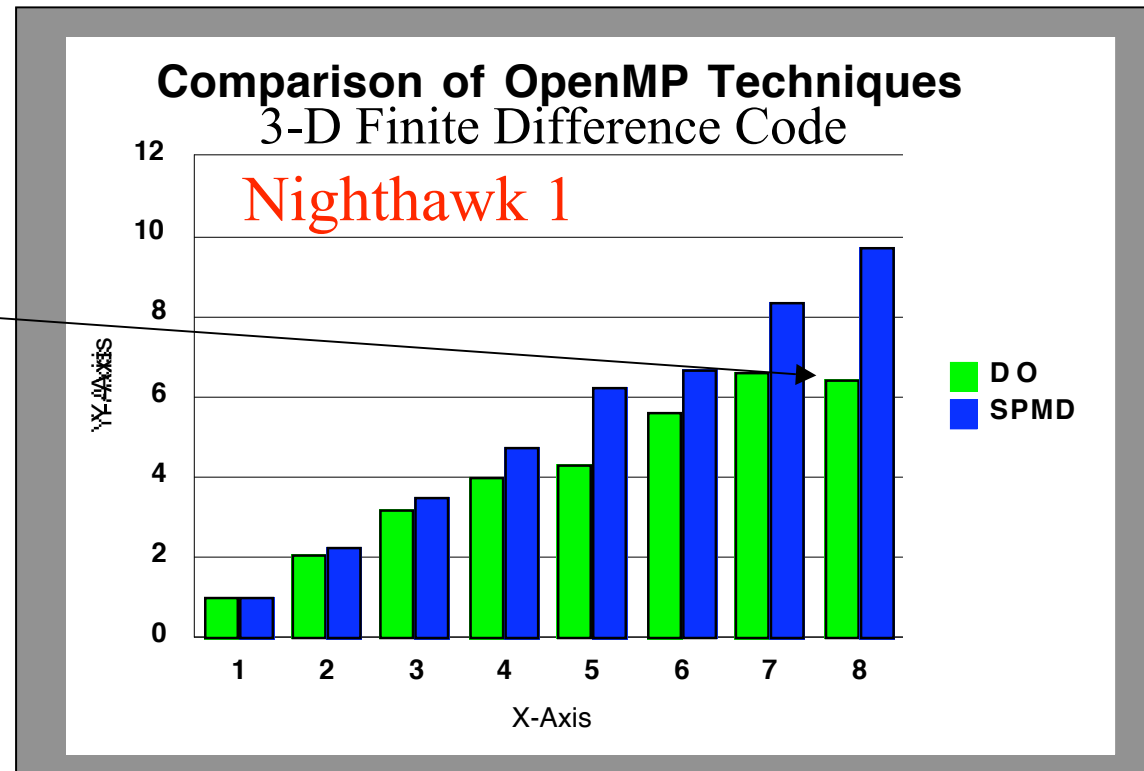
Etude au cas par cas (pas d'outils)

# Problèmes d'extensibilité

- Mode de programmation (grain)

J. Levesque (ACTC IBM), présentation a SCIComp 2K

Limite de l'extensibilité  
de l'approche « loop level  
parallelization » par rapport  
à l'approche SPMD.



- Limite Architecture/Système

De nombreux programmes OpenMP n'étaient pas extensibles sur l'Origin 2000 (migration de pages)

# Fonctions de bibliothèque

## Fonctions relatives aux verrous

- `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`,  
`omp_test_lock()`

## Fonctions de l'environnement d'exécution

- Modifier/vérifier le nombre de threads
  - `omp_set_num_threads()`, `omp_get_num_threads()`,  
`omp_get_thread_num()`, `omp_get_max_threads()`
- Autoriser ou pas l'imbrication des régions parallèles et l'**ajustement dynamique du nombre de threads dans les régions parallèles**
  - `omp_set_nested()`, **`omp_set_dynamic()`**, `omp_get_nested()`,  
`omp_get_dynamic()`
- Tester si le programme est actuellement dans une région parallèle
  - `omp_in_parallel()`
- Combien y a t-il de processeurs dans le système
  - `omp_num_procs()`

# Variables d'environnement

Contrôler comment les itérations de boucle sont ordonnancées.

`OMP_SCHEDULE "schedule[, chunk_size]"`

Positionner le nombre de threads par défaut.

`OMP_NUM_THREADS int_literal`

Indiquer si le programme peut utiliser un nombre variable de threads selon la région parallèle

`OMP_DYNAMIC TRUE || FALSE`

Indiquer si les régions parallèles imbriquées vont créer de nouvelles équipes de threads ou si les régions imbriquées seront sérialisées

`OMP_NESTED TRUE || FALSE`



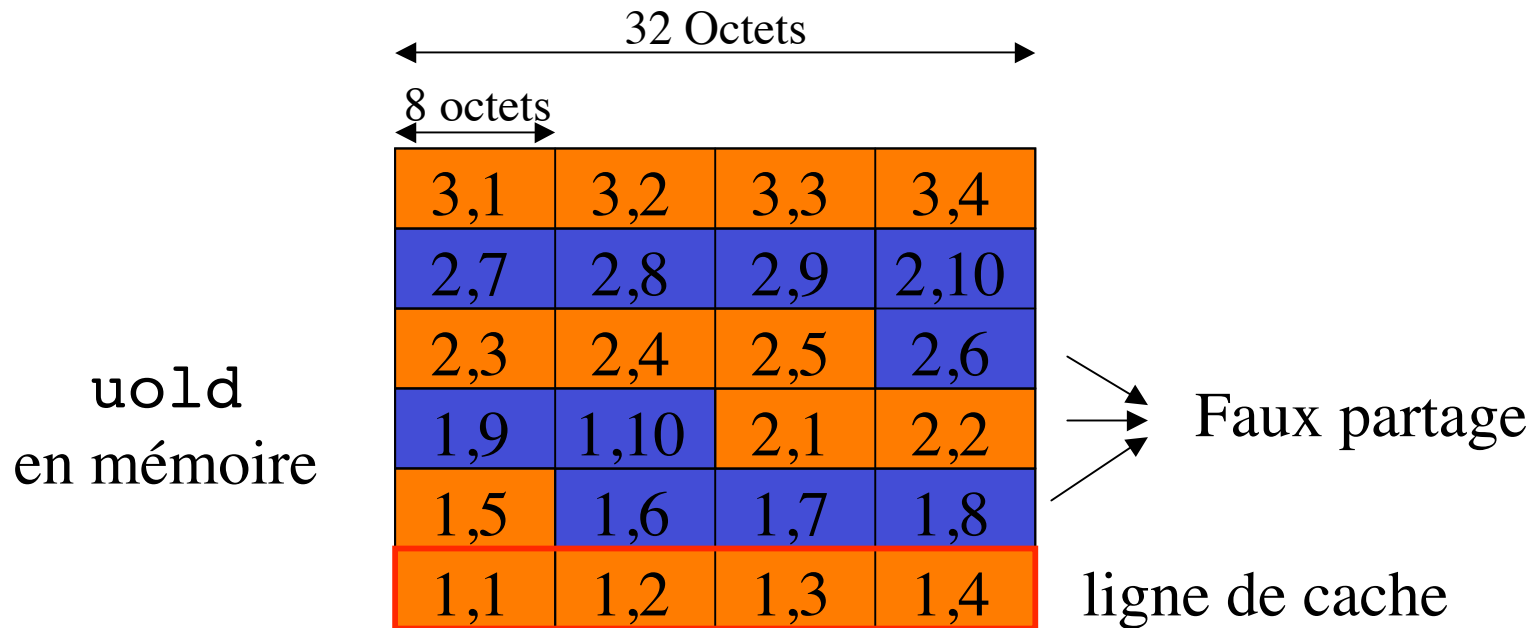
# Conclusion

- Modèle de programmation parallèle simple, portable.
- Attention aux performances
- OpenMP3 : orienté tâche
- Recherche : introduction des directives de distribution de données (entre autres)

# Une difficulté : éviter le false sharing

```
do j=1,10...  
omp parallel do schedule(static)  
    do i=1,10  
        uold(j,i) = u(j,i)  
    end do  
end parallel
```

Hypothèses : 2 threads  
u(10,10)  
u(x,y) -> 64 bits float



- ➔ 1) redimensionner les tableaux u et uold (calculs non pertinents)  
2) utiliser un chunk de 4 (moins de parallélisme)

# Le parallélisme imbriqué

- ❑ La notion de parallélisme imbriqué fait référence à la possibilité d'ouvrir une région parallèle à l'intérieur d'une région déjà parallèle
- ❑ Immédiatement, on peut voir les applications dans le cas de la parallélisation des nids de boucles, par exemple pour procéder 'par blocs'
- ❑ Toutefois, la plupart des compilateurs sérialisent les régions imbriquées, par manque de support au point de vue du système d'exploitation (en pratique, seules les NEC SX5 et les Compaq ES-40 permettent une telle approche)

# Problèmes de Binding

## SDSC Blue Horizon

A study on 1 node (SMP 8 procs)

Each OpenMP thread performs an independent matrix inversion

Results for OMP\_NUM\_THREADS=8

Without binding, threads migrate in about 15% of the runs

With thread binding turned on there was no migration

-> P=0.9876 probability overall results slowed by 25%  
for 144 nodes

**Slowdown occurs with/without binding**

Results for OMP\_NUM\_THREADS=7

12.5% reduction in computational power

**No threads showed a slowdown**

# OpenMP2

- Pas une transformation mais une amélioration
- Utilisation possible des constructeurs  
Workshare, Where et Forall de Fortran 90.
- Accepte les notations tableaux de Fortran 90
- Ajout de la clause `NUM_THREAD` à la directive  
`PARALLEL`
- La clause `REDUCTION` est étendue aux tableaux
- Routines de prise de temps (`OMP_GET_WTIME`)
- La spécification suggère le principe d'affinité des  
threads (deux régions parallèles successives devraient  
utiliser les mêmes threads avec les même numéro)