

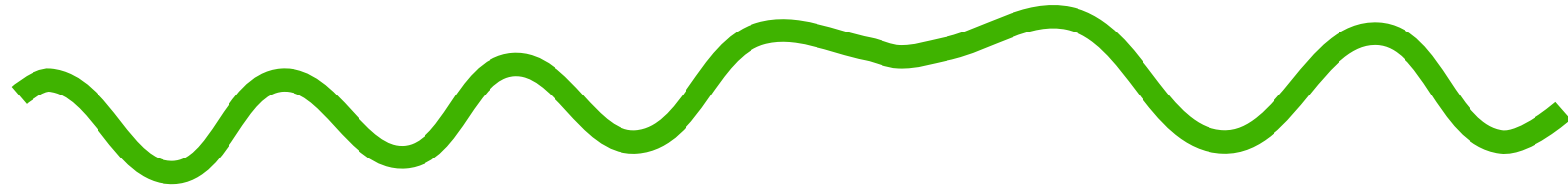
Introduction à OpenMP

***Outils pour le calcul scientifique à haute performance
École doctorale sciences pour l'ingénieur
mai 2001***

Pierre BOULET

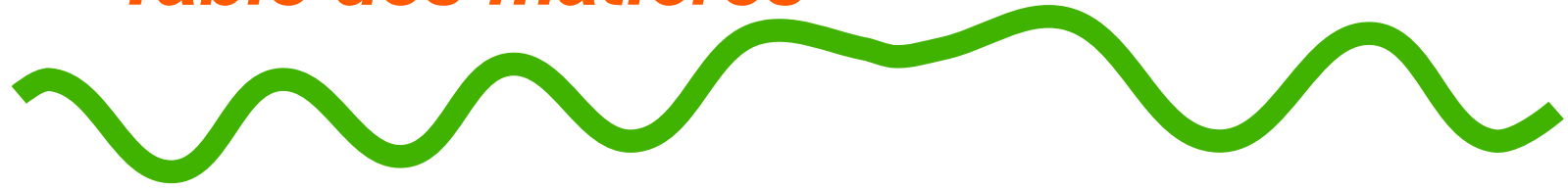
Pierre.Boulet@lifl.fr

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille



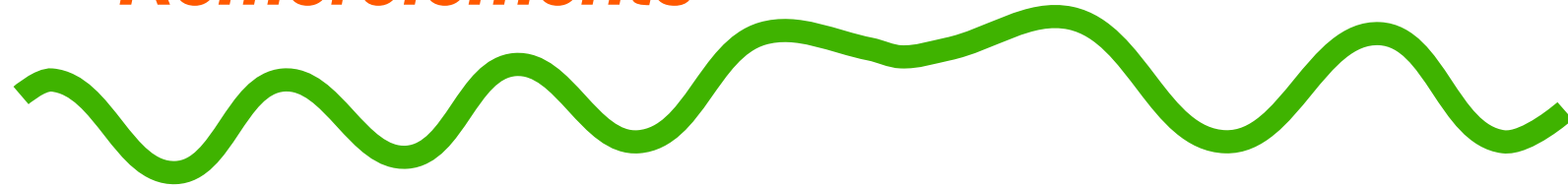
- ~ Ce cours est diffusé sous la licence GNU Free Documentation License,
<http://www.gnu.org/copyleft/fdl.html>
- ~ La dernière version de ce cours est accessible à partir de
<http://www.lifl.fr/west/courses/cshp/>
- ~ \$Id: openmp.tex,v 1.6 2002/03/18 07:18:21 marquet Exp \$

Table des matières



- ~ Modèle d'exécution
- ~ Partage du travail
- ~ Structuration des données
- ~ Synchronisation
- ~ Fonctions de bibliothèque / Variables d'environnement
- ~ Pour aller plus loin

Remerciements



Cette présentation d'OpenMP est essentiellement basée sur

~ *Tutoriel OpenMP à Supercomputing'99*

OpenMP Architecture Review Board

Tim Mattson, Rudolf Eigenmann

http://www.openmp.org/presentations/index.cgi?sc99_tutorial

~ *OpenMP Workshop*

Isabel Loebich

<http://www.hlrs.de/news-events/events/1999/openmp/openmp/>

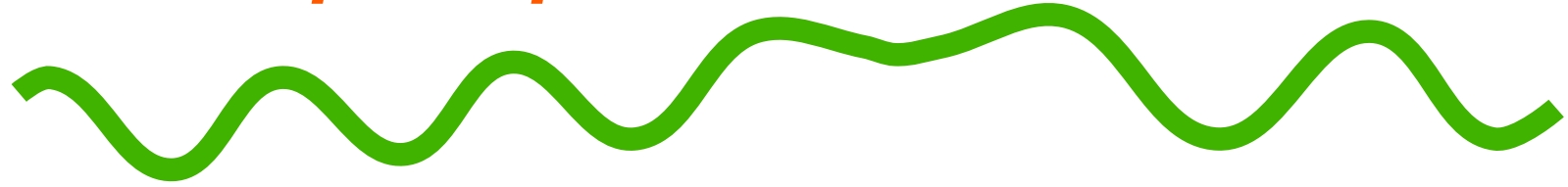
~ *Cours OpenMP*

IDRIS

[http://www.idris.fr/data/cours/parallel/
openmp/choix_doc.html](http://www.idris.fr/data/cours/parallel/openmp/choix_doc.html)

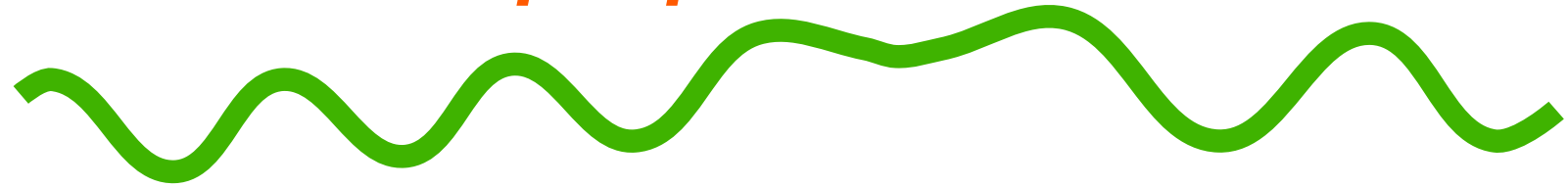


Pourquoi OpenMP ?

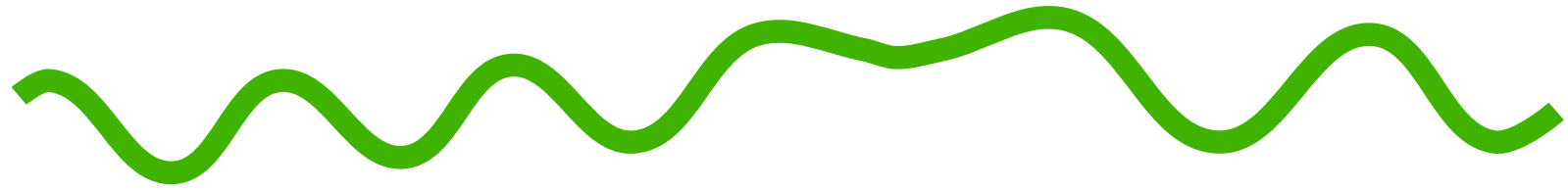


- ✓ les difficultés perçues de la programmation parallèle dépassent les avantages
 - ✓ diminuer les difficultés perçues
 - ✓ langage de haut niveau
- ✓ standardiser les pratiques de programmation à mémoire partagée
 - ✓ 15 ans de développement non concerté
 - ✓ recherche de la portabilité
- ✓ standard industriel contrôlé par l'**OpenMP Architecture Review Board**
 - ✓ organismes de recherche
 - ✓ constructeurs de supercalculateurs
 - ✓ fournisseurs de logiciels
 - ✓ sociétés de services

Qu'est-ce qu'OpenMP ?



- ~ API de programmation pour les applications multithreadées
 - ~ directives de compilation
 - ~ introduites par `!$OMP` en Fortran
 - ~ pragmas en C : `#pragma omp`
 - ~ bibliothèque de fonctions
 - ~ variables d'environnement
- ~ langages supportés :
 - ~ Fortran
 - ~ version 1.0 : octobre 1997
 - ~ version 1.1 : novembre 1999
 - ~ version 2.0 : novembre 2001
 - ~ C/C++
 - ~ version 1.0 : octobre 1998



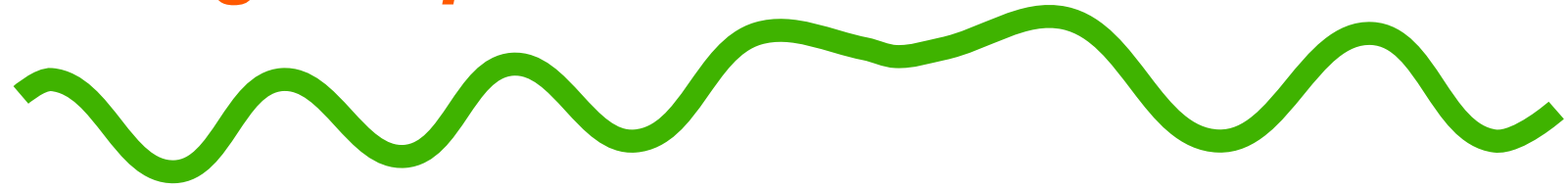
Modèle d'exécution

Modèle d'exécution



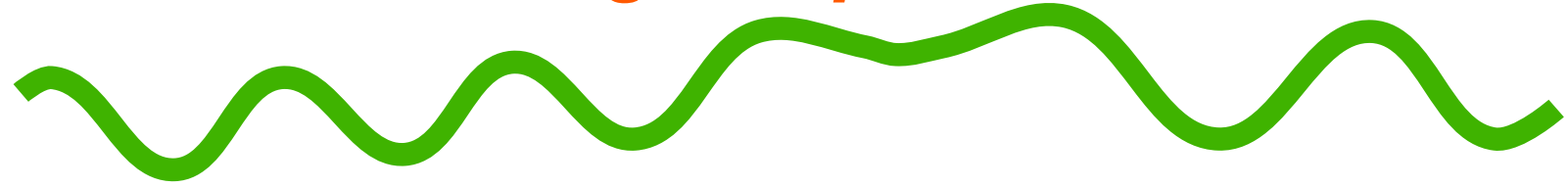
- ✓ langage de haut niveau
 - ✓ communications implicites
- ✓ modèle **fork/join**
 - ✓ un processus (partage de ressources)
 - ✓ contenant plusieurs processus légers (exécutions concurrentes)
 - ✓ succession de régions séquentielles et de régions parallèles
 - ✓ une tâche maître crée les autres
- ✓ parallélisme de contrôle
 - ✓ répartition des tâches
 - ✓ données partagées

Régions parallèles



- ✓ directive `omp parallel`
 - ✓ création de processus légers concurrents
 - ✓ aucune garantie du nombre de processeurs
 - ✓ réutilisation des processus légers dans plusieurs régions parallèles successives
 - ✓ exécution redondante du code sauf utilisation de directives de partage de travail
- ✓ terminaison par `omp end parallel`
 - ✓ synchronisation en fin de région parallèle
 - ✓ le maître continue l'exécution séquentielle
- ✓ combien de processus légers ?
 - ✓ déterminé par l'environnement d'exécution
 - ✓ numérotation à partir de 0 (maître)
 - ✓ on peut récupérer le numéro par `omp_get_thread_num()`

Portée des régions parallèles



- ✓ portée **lexicale**
 - ✓ le texte entre `omp parallel` et `omp end parallel`
 - ✓ doit être un bloc structuré (points d'entrée et de sortie uniques)
- ✓ portée **dynamique**
 - ✓ portée lexicale + toutes les fonctions et procédures appelées au sein de cette portée lexicale
- ✓ directives orphelines
 - ✓ partage du travail en dehors de la portée statique d'une région parallèle
 - ✓ exécution séquentielle ou parallèle en fonction du contexte d'exécution
 - ✓ permet une parallélisation incrémentale
- ✓ imbrication de régions parallèles
 - ✓ permise mais
 - ✓ le compilateur peut séquentialiser la région interne

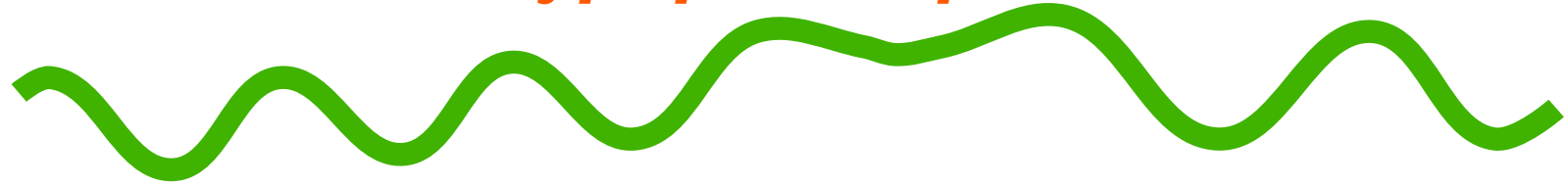
Exemples



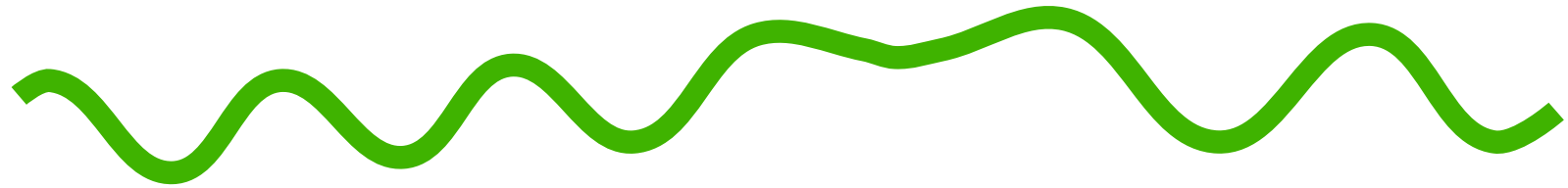
```
double A[1000];  
#pragma omp parallel  
{  
    int id;  
    id==omp_get_thread_num();  
    foo(id,A);  
}  
printf("Fin.\n");
```

```
!$OMP PARALLEL  
ID=OMP_GET_THREAD_NUM()  
PRINT *, 'Hello', ID  
PRINT *, 'world', ID  
!$OMP END PARALLEL
```

Utilisation typique d'OpenMP



- ✓ utilisation classique : parallélisation de boucles
 - ✓ trouver les boucles les plus coûteuses du programme
 - ✓ répartir les itérations entre des processus légers
- ✓ comment ces processus légers interagissent ils ?
 - ✓ mémoire partagée
- ✓ partage non intentionnel de données peut rendre l'exécution non déterministe
 - ✓ le résultat du programme dépend de l'ordonnancement des processus légers
- ✓ comment éviter ce non déterminisme ?
 - ✓ utiliser des synchronisations pour éviter les conflits de données
- ✓ synchronisations coûtent cher, donc
 - ✓ changer le stockage des données pour minimiser les besoins de synchronisation



Partage du travail

Parallélisme à gros grain

```
#pragma omp sections
{
    calcul_X();
#pragma omp section
    calcul_Y();
#pragma omp section
    calcul_Z();
}
```

```
!$OMP SECTIONS
!$OMP SECTION
CALL calcul_X()
!$OMP SECTION
CALL calcul_Y()
!$OMP SECTION
CALL calcul_Z()
!$OMP END SECTIONS
```

- chaque processus léger exécute un bloc structuré différent
- non extensible
- peut être combinée avec `omp parallel : omp parallel sections`

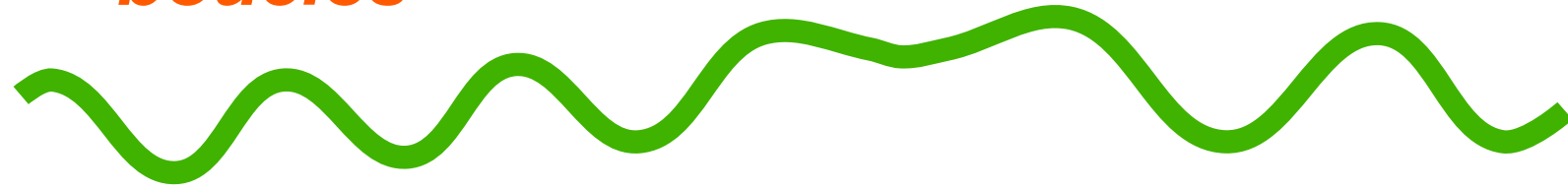
Parallélisation de boucles

```
#pragma omp for
for (i=0; i<n; i++){
    foo(i);
}
```

```
!$OMP DO
do i=0, n
    call foo(i)
end do
!$OMP END DO
```

- chaque processus léger n'exécute qu'une partie des itérations
- seule la boucle suivant immédiatement la directive est parallèle
- peut être combinée avec `omp parallel : omp parallel do`

Ordonnancement des itérations de boucles



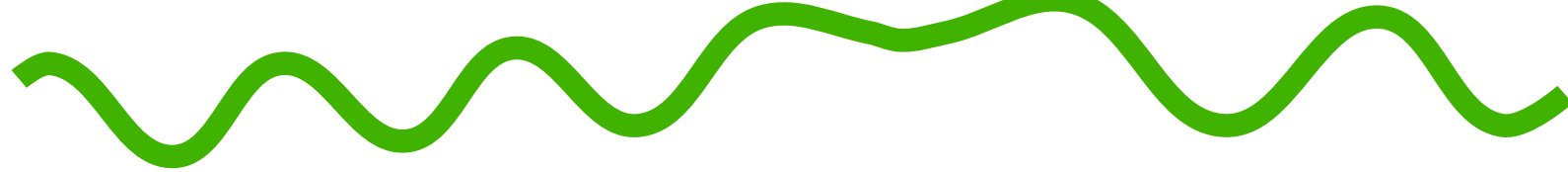
clause `schedule()` de la directive `do`

- ✓ `static(taille)` : répartition bloc/cyclique (défaut : bloc)
- ✓ `dynamic(taille)` : découpage en blocs (taille 1 par défaut) affectés selon la demande, bon pour l'équilibrage de charge
- ✓ `guided(taille)` : idem `dynamic` mais taille des paquets décroît exponentiellement (minimum `taille`), bon compromis équilibrage/surcoût
- ✓ `runtime` : choix reporté à l'exécution (variable d'environnement)

```
#pragma omp for schedule(static)
for (i=0; i<n; i++){a[i]=a[i]+b[i]}
```

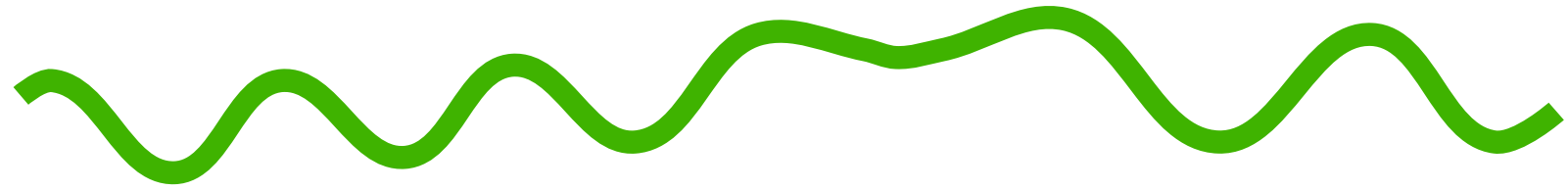
```
!$OMP DO SCHEDULE GUIDED
do i=0, n
  call foo(i)
end do
```

master : sur le thread maître



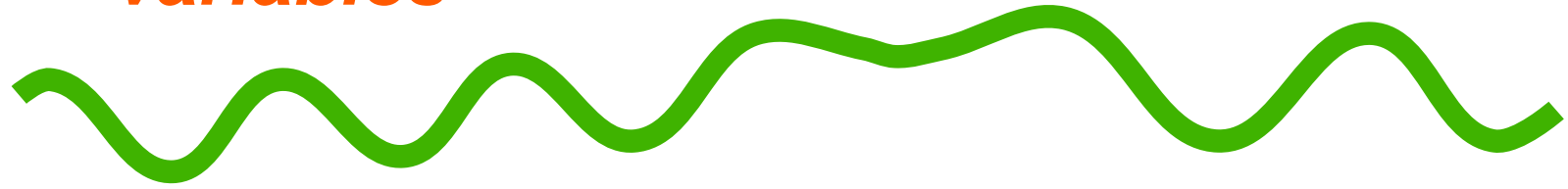
- une région `master` est exécutée seulement sur le processus léger maître
- rappel : par défaut, réplication du code
- exemple :

```
!$OMP PARALLEL
...          !--- region repliquee
!$OMP MASTER
  call sauve()!--- maitre seulement
!$OMP END MASTER
...          !--- region repliquee
!$OMP END PARALLEL
```



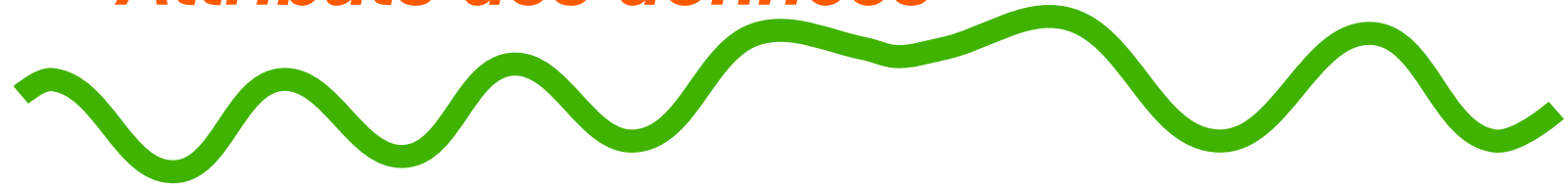
Structuration des données

Rappel : les différents types de variables



- ✓ statiques / dynamiques
 - ✓ **statique** : emplacement mémoire défini à la compilation
 - ✓ **dynamique** : emplacement mémoire alloué à l'exécution
- ✓ globales / locales
 - ✓ **globales** : durée de vie = tout le programme
 - ✓ **locales** : durée de vie = un bloc structuré
- ✓ variables et processus légers
 - ✓ variables statiques partagées
 - ✓ variables dynamiques locales à chaque processus léger

Attributs des données



- ✓ données partagées : clause `shared`
 - ✓ visibles par tous les processus légers
 - ✓ par défaut les variables globales statiques
 - ✓ s'applique aux directives de régions parallèles
- ✓ données privées : clause `private`
 - ✓ répliquées sur chaque processus léger
 - ✓ indéfinies à l'entrée de la région parallèle
 - ✓ valeurs non transmises à la région séquentielle qui suit
 - ✓ par défaut les variables locales et les variables globales dynamiques
 - ✓ s'applique aux directives de régions parallèles et de partage du travail

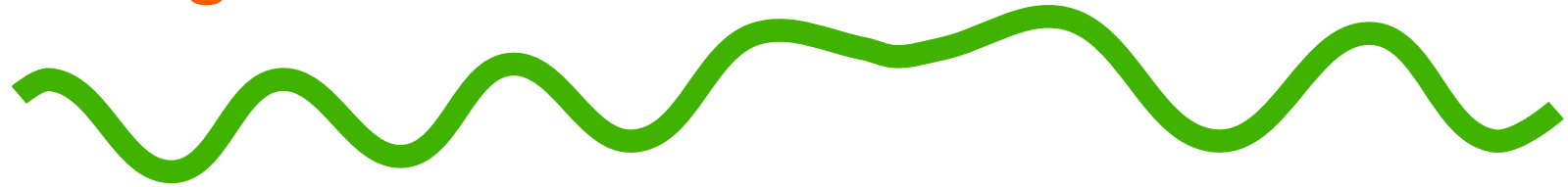
Exemples

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
IAM = OMP_GET_THREAD_NUM()
NP = OMP_GET_NUM_THREADS()
IPOINTS = NPOINTS/NP
CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

```
IS = 0
!$OMP PARALLEL DO PRIVATE(IS)
DO J=1,100
  IS = IS + 1
END DO
PRINT *, IS
```

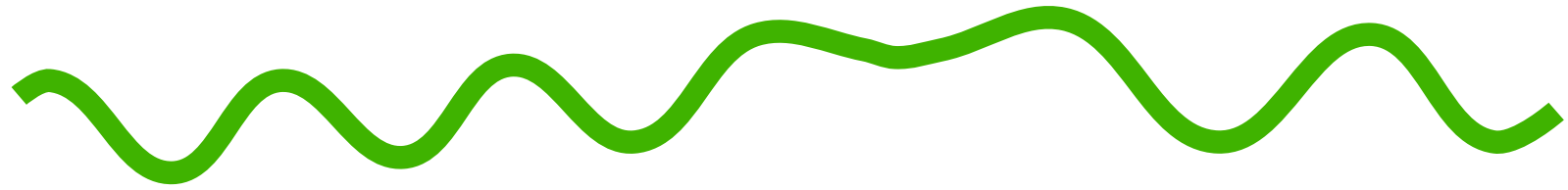
- ~ IS non initialisé dans la boucle
- ~ après la boucle : valeur indéterminée malgré l'initialisation

Transmission de valeurs entre régions



- ✓ `firstprivate`
 - ✓ cas particulier de `private`
 - ✓ initialise chaque copie privée par la valeur du maître
- ✓ `lastprivate`
 - ✓ uniquement pour les boucles parallèles
 - ✓ la valeur de la dernière itération de la boucle parallèle est copiée dans une variable globale
- ✓ code corrigé :

```
IS = 0
!$OMP PARALLEL DO FIRSTPRIVATE(IS) &
!$OMP LASTPRIVATE(IS)
DO J=1,100
    IS = IS + 1
END DO
PRINT *, IS
```



Synchronisation

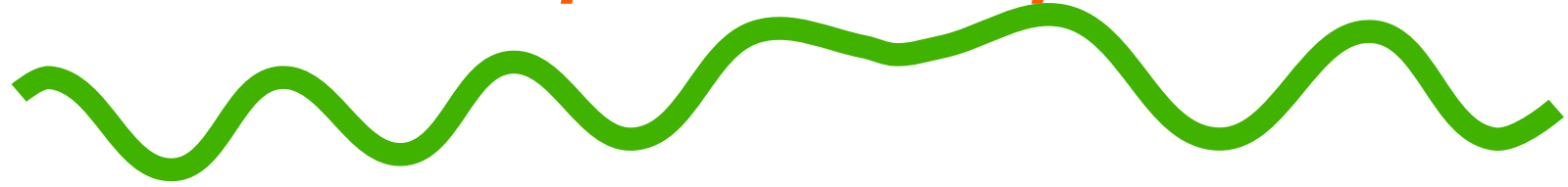
single



- ~ une région `single` est exécutée par un seul processus léger
 - ~ en général le premier arrivé
- ~ barrière de synchronisation à la fin

```
~ #pragma omp parallel private(tmp)
{
    debut_travail();
    #pragma omp single
    {échange_frontieres();}
    suite_travail();
}
```

Barrières explicites et implicites



~ barrières de synchronisation implicites

~ `end parallel`

~ `end do`

~ `end sections`

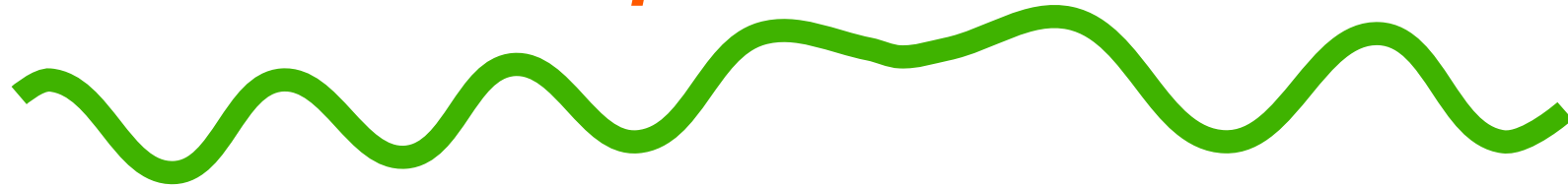
~ `end critical`

~ `end single`

~ explicite : directive `omp barrier`

~ suppression des synchronisations implicites par la clause `nowait`

Sections critiques



~ un seul processus léger à la fois dans une section critique

```
!$OMP PARALLEL DO PRIVATE(B) SHARED(RES)
DO I=1,NITERS
  B=TRAVAIL(I)
  !$OMP CRITICAL
  CALL CONSOMME(B,RES)
  !$OMP END CRITICAL
END DO
```

~ cas particulier des instructions `atomic`

~ ne s'applique qu'à des mises à jour de cases mémoires

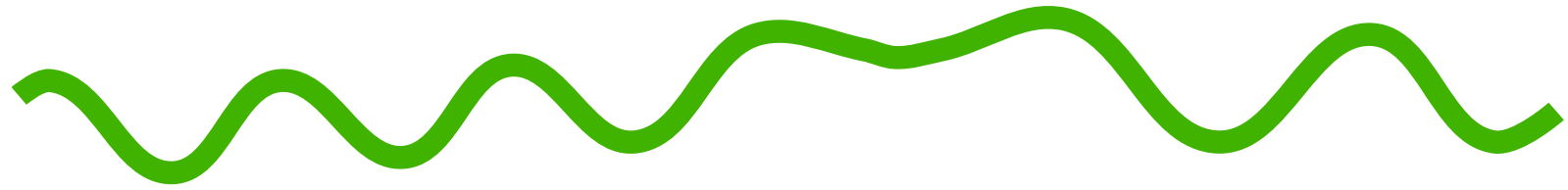
```
#pragma omp parallel private(b)
{ b=foo(i);
#pragma omp atomic
  x=x+b; }
```

Réductions



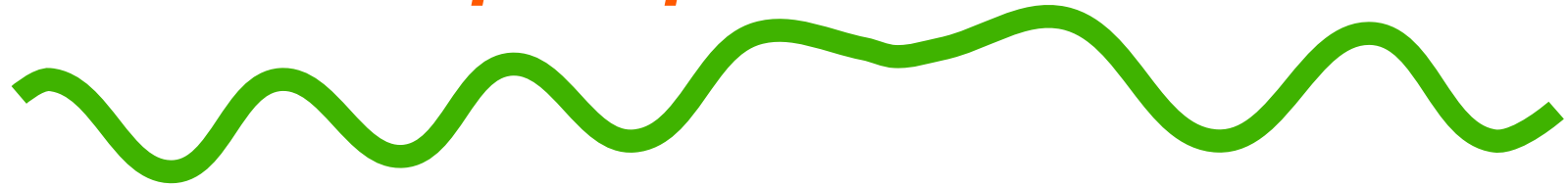
- ~ clause `reduction(op : liste)`
 - ~ les variables dans la `liste` doivent être partagées dans la région englobante
 - ~ à l'intérieur de la région :
 - ~ initialisation par l'élément neutre
 - ~ traitement sur la copie locale
 - ~ réduction à la sortie

```
#pragma omp parallel for reduction(+:r) private(t)
for (i=0; i<1000; i++){
    t=foo(i);
    r=r+t
}
```



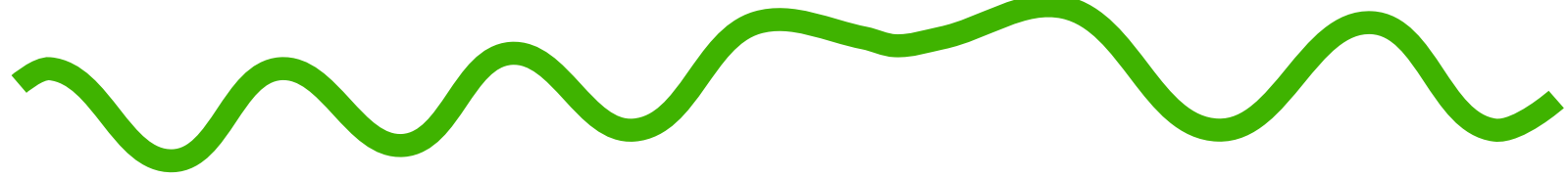
Fonctions de bibliothèque / Variables d'environnement

Bibliothèque OpenMP

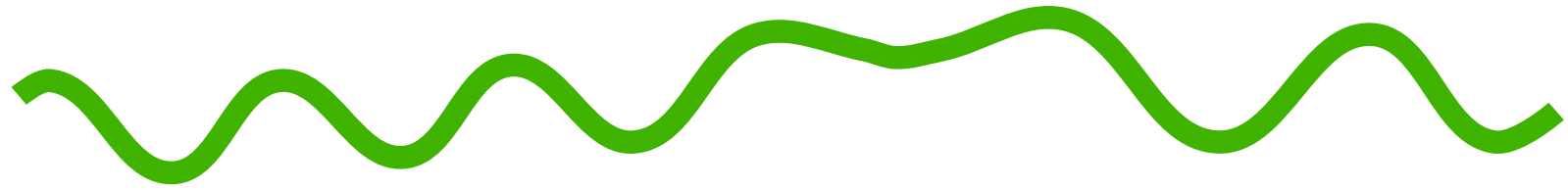


- ✓ gestion de verrous
- ✓ gestion de l'environnement d'exécution
 - ✓ modification/vérification du nombre de processus légers
 - ✓ test de région parallèle
 - ✓ nombre de processeurs dans le système
 - ✓ dynamicité du nombre de processus légers
 - ✓ création de nouveaux processus légers lors de l'imbrication de régions parallèles
- ✓ priorité par rapport aux variables d'environnement mais pas par rapport aux directives

Variables d'environnement

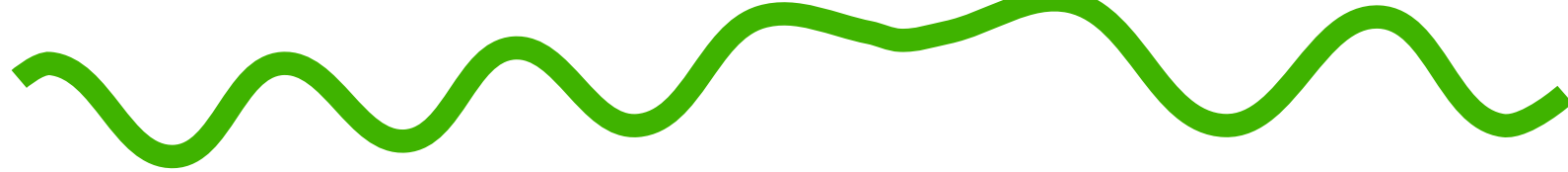


- ~ contrôle de l'ordonnancement « runtime »
 - ~ OMP_SCHEDULE
- ~ nombre de processus légers à utiliser
 - ~ OMP_NUM_THREADS
- ~ dynamique du nombre de processus légers
 - ~ OMP_DYNAMIC
- ~ création de nouveaux processus légers lors de l'imbrication de régions parallèles
 - ~ OMP_NESTED



Pour aller plus loin

Parallélisation du calcul de PI



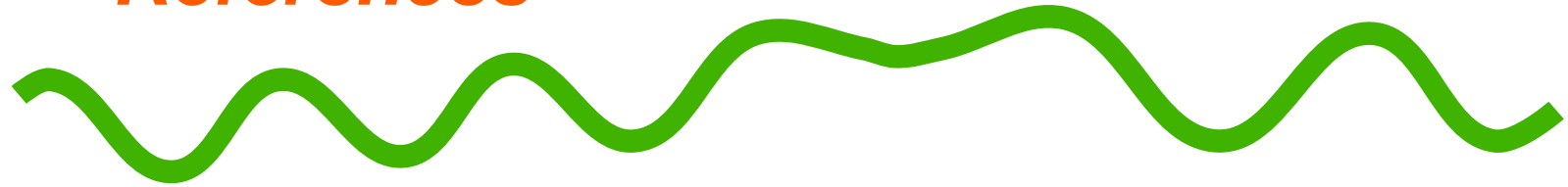
- ✓ C ou Fortran au choix
- ✓ voir `~marquet/pi-openmp/` sur bigblue
- ✓ versions successives
 - ✓ version séquentielle (`pi`)
 - ✓ parallélisation de la boucle (`pi1`)
 - ✓ structuration des données (`pi2`)
 - ✓ gestion des conflits avec section critique (`pic`)
 - ✓ localisation du calcul (`pic2`)
 - ✓ utilisation d'une réduction (`picrc`)

Ce qu'on a passé sous silence



- ✓ partage du travail
 - ✓ règles d'imbrication
 - ✓ directive `workshare` (OpenMP 2.0) pour paralléliser les `forall` et expressions de tableaux Fortran 90
- ✓ structuration des données
 - ✓ clause `threadprivate`
- ✓ synchronisation
 - ✓ directive `flush`
 - ✓ gestion des verrous par les fonctions de bibliothèque

Références



~ le site de l'OpenMP ARB : <http://www.openmp.org/>

~ standards

~ tutoriels

~ exemples de code

~ support de cours de l'IDRIS : <http://www.idris.fr/>

~ documentation IBM :

<http://www.ibm.com/software/ad/caix/>

<http://www.ibm.com/software/ad/fortran/>

[xlfortran/library.html](http://www.ibm.com/software/ad/fortran/xlfortran/library.html)