Cours OpenMP

CNRS - IDRIS

Version 1.3

Dernière mise à jour le 1 septembre 2000

Etienne Gondet : gondet@idris.fr
Pierre-Francois Lavallée : lavallee@idris.fr

INSTITUT DU DEVELOPPEMENT
ET DES RESSOURCES
EN INFORMATIQUE SCIENTIFIQUE

Préface

Afin d'obtenir des renseignements complémentaires, voici la liste des services qui pourront vous aider ainsi que leurs adresses électroniques :

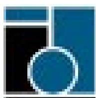
Assistance IDRIS : assist@idris.fr

Documentation IDRIS : doc@idris.fr

Secrétariat IDRIS : secretariat@idris.fr

Cours réalisé par Etienne Gondet : gondet@idris.fr

Copyright © Reproduction totale interdite sans autorisation de l'auteur. Reproduction partielle autorisée pour l'usage du copiste. Ce document est basé sur les *drafts* de 1997 à 2000 d'OpenMP qui est une propriété intellectuelle de l'ARB (*Architecture Review Board).*



Avant Propos

OpenMP amène aujourd'hui une interface standard de haut niveau pour une programmation parallèle de type SPMD[1] sur machine à mémoire partagée ou au moins virtuellement partagée[2]. Basée sur les techniques du *multithreading,* on peut considérer OpenMP comme l'un des grands standards au service du calcul scientifique.

Public visé

Ce cours qui aborde de manière progressive et systématique les différentes composantes d'OpenMP s'adresse à deux types d'auditeurs ; tout d'abord à de réels débutants en tant que manuel de référence le plus complet possible, ensuite à un public de faux débutants se heurtant à des difficultés conceptuelles au-delà des aspects purement syntaxiques. Ainsi, plus qu'un simple catalogue descriptif, en laissant la primauté à une approche conceptuelle, j'ai essayé d'offrir un réel guide succeptible de proposer un choix critique parmi les diverses constructions qu'offrent OpenMP. Précisons également que ce livre traite peu de sujets tels que les techniques et outils de déverminage ou d'optimisation; Thèmes qui nécessitent un cours à part entière.

1. SPMD : Single Program Multiple Data
2. telle que la SGI O200

Comment utiliser ce livre

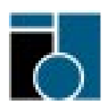
Après quelques généralités, afin de mieux situer OpenMP parmi les autres types de parallélisation existante (*Data-Parallelism* ou *Message Passing*), les chapitres 2, 3, 4 et 5 présentent les différents aspects d'OpenMP. Signalons toutefois que le chapitre 3 met l'accent sur la localisation des données en mémoire, point qui recèle la plupart des pièges relatifs à ce type de programmation parallèle. Le chapitre 6 est plus spécialement voué aux concepts sous-jacents du standard OpenMP (ordonnancement, *orphaning, nesting*) et à l'illustration de ceux-ci par des exemples plus concrets. Afin d'être éviter en première lecture, les sous-chapitres et paragraphes facultatifs ou plus ardus ont été marqués d'un astérisque * ; quant à ceux marqués d'un double astérisque **, ils sont relatifs aux nouveautés de la version 2 d'OpenMP encore présentés de façon très théorique et peuvent donc également être contourné lors d'une première approche. En effet OpenMP 2 n'est encore qu'en phase finale de validation et à cette date, il n'existe pas d'implémentations de ces nouvelles fonctionnalités.

Les exemples ont tous été testés sur au moins 3 plates-formes (compilateur : *f90* sur NEC-SX5, *f90* sur SGI-O2000, *xlf90_r*[1] sur IBM Night-Hawk 1)[2]. Précisons à l'attention des développeurs C/C++ que même si ce cours s'adresse plus spécialement à des développeurs FORTRAN, les syntaxes et concepts du *draft* pour les langages C/C++ sont extrêmement similaires; En effet le standard OpenMP peut être envisagé pour n'importe quel langage de programmation tout en gardant les mêmes concepts intrinsèques.

1. version ré-entrante du compilateur FORTRAN 90 d'IBM.
2. à l'exception de ceux concernant des fonctionnalités propres à la version 2 d'OpenMP

A l'attention des programmeurs déjà expérimentés avec les interfaces propriétaires plus anciennes qu'OpenMP.

Bien qu'OpenMP ait été, avant tout, un effort de synthèse des constructions propres aux différents ensembles de directives de *microtasking* existant depuis une dizaine d'années, la "norme" OpenMP est abordée dans cet ouvrage "telle qu'en elle même" afin de ne pas dérouter par des références historiques systématiques à des interfaces souvent propriétaires et donc spécifiques à quelques constructeur. D'autre part, on ne soulignera jamais assez qu' OpenMP va sensiblement au delà de ses prédecesseurs en intégrant la possibilité de mise en oeuvre des techniques de décomposition de domaines. Ainsi OpenMP a cette double richesse d'avoir été d'emblée pensé et conçu pour permettre la parallélisation à gros grain (*Coarse Grain*) basée sur les techniques de décomposition de domaine) ou à grain fin (*fine grain*) basée sur les techniques dites de *microtasking*.



OpenMP

1 - INTRODUCTION 14

1.1 Bibliographie 15

1.2 OpenMP : un nouveau standard!. 16

1.2.1 A qui OpenMP s'adresse t'il? 16

1.2.2 La petite histoire d'OpenMP * 17

1.2.3 Les acteurs de la standardisation 18

1.3 Situer OpenMP parmi d'autre modèle de parallélisation . . 20

1.3.1 Communications implicites ou explicites. 20

1.3.2 Les architectures adaptées 21

1.4 Structures d'OpenMP 22

1.4.1 L'architecture générale 22

1.4.2 Constructions OpenMP 24

1.5 Terminologie 25

2 - PRINCIPES GENERAUX 26

2.1 Compilation, chargement et exécution. 27

2.2 Modèle d'exécution 28

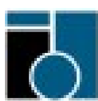
2.2.1 Le modèle monoprocessus 28

2.2.2 La construction mère : la région parallèle 29

2.2.3 La numérotation des tâches 31

2.2.4 La portée d'une région parallèle 31

2.2.4.1 La portée lexicale d'une région parallèle 31



2.2.4.2 La portée dynamique 31

2.3 Clauses de la directive PARALLEL 32

2.4 Les constructions OpenMP *. 33

2.4.1 La portée locale. 33

2.4.2 La portée lexicale 34

2.4.3 La portée dynamique 34

2.5 Règles syntaxiques des directives 35

2.5.1 Sentinelles et directives. 35

2.5.2 Directives : lignes suite 36

2.5.3 Compilation conditionnelle par sentinelle ou macro . . 36

2.5.3.1 Par une sentinelle : !$ 37

2.5.3.2 Par une macro : _OPENMP 37

3 - STRUCTURATION DES DONNEES 38

3.1 Gestion mémoire et processus légers 39

3.1.1 Les différents types de variables.. 39

3.1.1.1 Variables statiques ou automatiques ? 39

3.1.1.2 Variables globales ou locales 39

3.1.1.3 Variables locales automatiques 40

3.1.1.4 Variables locales rémanentes 41

3.1.1.5 Les tableaux automatiques 41

3.1.1.6 Les tableaux dynamiques. 42

3.1.2 Les différentes zones mémoires 43

3.1.2.1 La zone U. 43

3.1.2.2 La zone TXT 43

OpenMP

3.1.2.3 La zone DATA 43

3.1.2.4 La zone BSS. 43

3.1.2.5 La pile (*stack*) et le tas (*heap*) 44

3.1.3 Quelles données pour quelle zone mémoire ? 44

3.1.3.1 Cas des variables locales automatiques ? 45

3.1.4 Les tâches OpenMP : des processus légers 46

3.2 Attribut des données 48

3.2.1 Les données partagées : clause SHARED 48

3.2.2 Les données privées : clause PRIVATE 48

3.2.3 Clause FIRSTPRIVATE 50

3.2.4 Clause LASTPRIVATE 50

3.2.5 Restrictions sur FIRST et LASTPRIVATE 50

3.3 La clause DEFAULT( ) 51

3.3.1 DEFAULT(SHARED) 51

3.3.2 DEFAULT(PRIVATE) 51

3.3.3 DEFAULT(NONE) 51

3.4 La directive THREADPRIVATE 52

3.4.1 En OpenMP 1 52

3.4.2 Clause COPYIN des régions parallèles 53

3.4.3 Exemple de THREADPRIVATE + COPYIN. 54

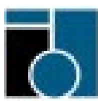
3.5 Les tableaux dynamiques 56

3.6 Les modules Fortran 95 * 57

3.6.1 Extension de **THREADPRIVATE et COPYIN** ** 59

3.7 Statut implicite des variables. 60

3.8 Exercice récapitulatif. 61



4 - PARTAGE DU TRAVAIL 62

4.1 La directive SECTIONS. 63

4.2 La directive DO 64

4.2.1 Clause LASTPRIVATE 66

4.3 La directive WORKSHARE ** 67

4.4 La directive PARALLEL SECTIONS 69

4.5 La directive PARALLEL DO. 70

4.6 La directive PARALLEL WORKSHARE. 71

4.7 La directive MASTER 72

5 - SYNCHRONISATIONS 74

5.1 Utilité 75

5.2 La directive SINGLE. 76

5.2.1 La clause COPYPRIVATE ** 77

5.3 Les barrières 78

5.3.1 Les barrières explicites 78

5.3.2 Les barrières implicites. 78

5.4 Les zones critiques : directive CRITICAL 79

5.5 La mise à jour atomique : !$OMP ATOMIC 80

5.6 La clause REDUCTION 82

5.7 Clause ORDERED de DO et directive ORDERED 84

5.8 La directive FLUSH * 86

5.8.1 Le vidage implicite de *buffers* *86*

5.8.2 Le vidage explicite de *buffers*. *87*

5.9 Exercice récapitulatif 89

6 - CONCEPTS ET EXEMPLES 90

6.1 OpenMP et Fortran 95 * 91

6.1.1 Limitations OpenMP 1 levées avec OpenMP 2 91

6.1.2 Restrictions explicites en OpenMP 1 et 2. 92

6.2 Règles de détermination du statut des variables. 93

6.3 Ordonnancement d'une boucle partagée. 94

6.3.1 Le mode statique 94

6.3.2 Le mode dynamique 95

6.3.3 Le mode *guided* *97*

6.3.4 Spécificités du *chunk* indépendant du mode 99

6.4 Régions parallèles dynamiques ou non *. 100

6.5 Niveaux de répartition du travail.. 102

6.6 L'*orphaning*. *103*

6.7 Le *nesting* * 104

6.8 *Binding* * 105

6.9 *Loop-level parallelism* *106*

6.10 Décomposition de domaines. 108

6.10.1 Méthode de décomposition de domaines 108

6.10.2 Méthodologie et performances. 109

6.11 Réflexions sur les performances 110

6.11.1 Règles indépendantes des architectures. 110

6.11.2 Risques de dégradation selon la plate-forme utilisée . 111

7 - BIBLIOTHEQUES ET VARIABLES................................ 114

7.1 Les variables d'environnement. 115

OpenMP

7.2 La *Run-time Library* d'OpenMP 116

7.2.1 Généralités. 116

7.2.2 Les sous-programmes de *timing* ** 117

7.2.2.1 Le *timer* standard OpenMP 2 117

7.2.2.2 Précision du *timing* OpenMP 117

7.2.3 Les sous-programmes relatifs au contexte 118

7.2.3.1 Fixer ou connaître le nombre de *threads* *118*

7.2.3.2 Connaître mon numéro de tâches 118

7.2.3.3 "To be or not to be in // region !". 119

7.2.3.4 Nombre maximum de *threads* *119*

7.2.3.5 Nombre de processeurs 119

7.2.3.6 Le mode dynamique des régions parallèles. . . . 120

7.2.3.7 *Nesting or not nesting* ? 120

8 - CONCLUSIONS 122

8.1 Méthodologie d' "OpenMPisation" 123

8.1.1 Approche intégrée 123

8.1.2 Approche progressive 124

8.2 Les atouts d'OpenMP 125

8.3 Limitations du parallélisme via OpenMP 126

8.3.1 Lois générales sur l'accélération d'un code parallèle . . 126

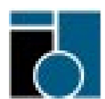
8.3.2 Limitations propres à OpenMP 128

8.4 Outils et bibliothèques tierces. 129

8.4.1 Les implémentations libres d'OpenMP. 129

8.4.2 Compilateurs et outils correspondants 129

8.4.3 Librairies parallèles compatibles avec OpenMP. 130



8.5 Evolutions d'OpenMP 131

8.5.1 OpenMP 2 (**) 131

8.5.2 Evolutions à plus long terme, ... 132

ANNEXES 134

A -synoptique OpenMP 2.0 135

B -Directives C/C++ 136

C -Verrous * 137

D -Comportement dépendant des implémentations * 139

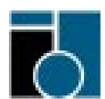
E -Exemple de travail NQS sur NEC SX5 141

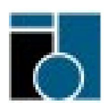
F -Passer du *macrotasking* à OpenMP * 142

G -Lexique général 143

H -Lexique OpenMP * 144

I -Exercices récapitulatifs, corrections 146





1 - INTRODUCTION

OpenMP

1.1 Bibliographie

A ce jour, il n'existe aucun livre sur OpenMP en anglais ou en français. On trouve nénamoins de nombreux supports sur la toile.

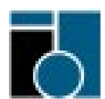
Sur OpenMP :

- ☞ adresse du forum OpenMP : http://**www.openmp.org**
  - ☚"brouillon"[1] du standard OpenMP validé par l'ARB,
- ☞ **tutoriaux** très complets de *supercomputing* 98 et 99 :
  - ☚**http://www.openmp.org/presentations**
- ☞ tutorial Compaq/Dec
  - ☚http://www.digital.com/hpc/ref/index_slides_13.html
- ☞ *micro-bench*[2] pour évaluation d'une implémentation.
  - ☚www.epcc.ed.ac.uk/research/openmpbench/dowload.html

Sur Fortran :

- ☞ LIGNELET P., *Manuel complet du langage **Fortran 90** et **Fortran 95**,* calcul intensif et génie logiciel, col. Mesures physiques, MASSON, 1996, (320 pages), ISBN 2-225-85229-4.
- ☞ ELLIS T.M.R., PHILLIPS I.R., LAHEY T., *Fortran 90 programming,* ADDISON WESLEY*, 1994, (825 pages),* ISBN 2-201-54446-4.
- ☞ **Fortran 95** Handbook, MIT Press, 1997, (711 pages), ISBN 0-07-000406-4.
- ☞ CORDE P. & DELOUIS H., Cours Fortran 95, IDRIS[3]/CNRS, 2000, (250 pages).
  - ☚http://www.idris.fr/data/cours/lang/f90/choix_doc.html

1. "brouillon" étant ici la traduction litigieuse de draft en anglais
2. De l'EPCC (Edinburgh Parallel Computer Center).
3. IDRIS Institut de développement Des Ressources en Informatique Scientifique du CNRS.



1.2 OpenMP : un nouveau standard!

1.2.1 A qui OpenMP s'adresse t'il?

Il s'agit d'un ensemble de procédures et de directives de compilation identifiées par un mot clef initial !$OMP visant à réduire le temps de restitution lors de l'éxécution d'un programme sans en changer la sémantique.

- ☞ Standard "**industriel**" basé sur une **API** pour
  - la **portabilité** d'applications ***multithreadés***,
  - sur machines à mémoire **partagée** ou à mémoire **virtuellement partagée.**
- ☞ Standardisation de la parallélisation
  - à grain fin (sur des boucles),
  - à gros grain (exemple : décomposition de domaines).
- ☞ Pour les langages
  - **Fortran 90** : version 1.0 en Octobre 1997, version 1.1 en Novembre 1999 incluant les interprétations de l'ARB[1], version 2.0 vers Octobre-Novembre 2000.
  - **C** et **C++** : version 1.0 en Octobre 1998.
- ☞ Largement implémenté aujourd'hui sur
  - sur noeuds multiprocesseurs à mémoire partagée (SMP : Shared Memory Processors); Compaq, Cray PVP, IBM, HP, NEC SX5, SGI ORIGIN, SUN, noeuds *beowulf*, ...
  - pas sur architectures distribuées (Cray-T3E ou VPP 5000).

1. l'ARB, Architecture Review Board est l'organisme propriétaire de la marque OpenMP.

OpenMP

1.2.2 La petite histoire d'OpenMP *

☞ A l'origine, le comité X3 de l'ANSI autorisa un sous-comité X3H5 à reprendre le travail du FORUM PCF (Parallel Computing Forum) qui était un groupement informel d' industriels afin de standardiser un parallélisme de contrôle sur les boucles DO.

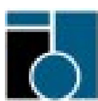
- ☛ Néanmoins, ils parvinrent à produire un *draft* qui ne fut jamais finalisé.
- ☛ L'apparition d'architectures à mémoire distribuée et de bibliothèques de *message passing* (PVM-MPI) amenèrent à l'époque à penser que la parallélisation sur mémoire partagée était devenue obsolète.
- ☛ Chaque constructeur définit son propre jeu de directives.

☞ A partir de 1996-97, eut lieu un revirement pour deux raisons majeures :

- ☛ un retour en grâce des architectures à mémoire partagée chez beaucoup de constructeurs,
- ☛ certains acteurs industriels considérant la parallélisation par *message passing* longue et fastidieuse, manifestèrent leur intérêt pour un modèle de parallélisation peut être moins universel mais nécessitant moins d'efforts.

☞ OpenMP résulta d'un large agrément entre usagers et acteurs industriels en tant que standard "industriel" (et non comme standard issu d'un comité de normalisation).

☞ Cependant à la différence de la majorité de ses prédécesseurs, OpenMP supporte également la parallélisation à gros grain (coarse-grain) par décomposition de domaines notamment.



1.2.3 Les acteurs de la standardisation

Une des caractéristiques du processus de standardisation a été de tenir compte des différents types d'acteurs industriels dans le petit monde du HPC[1] (même s'ils sont pour la plupart et une fois de plus américains).

Ainsi, la présence d'organisme de recherche demeura faible tandis que les concepteurs de lo(pro)giciels furent bien représenté en sus des constructeurs informatiques. C'est ce qui explique l'approche très pragmatique qui a prévalu à la conception d'OpenMP afin d'éviter de définir des constructions trop complexes à réaliser ou trop abstraite.

- ☞ Les Organisations de la recherche
  - ✒ Purdue University.
  - ✒ US Department of Energy ASCI Program.

- ☞ Les constructeurs informatiques
  - ✒ Compaq Computer Corp.
  - ✒ Hewlett-Packard Company.
  - ✒ Intel Corp.
  - ✒ SGI (Silicon Graphics Incorporate)..
  - ✒ CRI (Cray Research).
  - ✒ Sun Microsystems.
  - ✒ International business machines.

1. High Performance Computing.

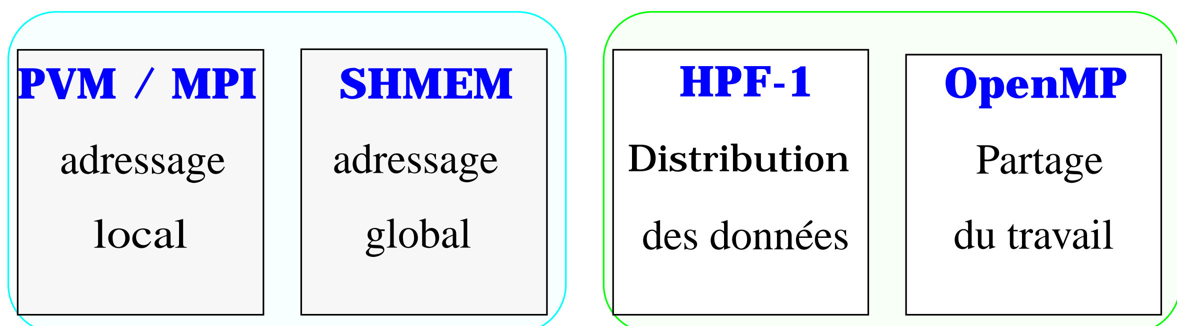
- Des fournisseurs de logiciels
  - Absoft Corp.
  - Edinburgh Portable Compilers.
  - Kuck & Associates, Inc.
  - Myrias Computer Technologies.
  - NAG (Numerical Algorithms Group Ltd.)
  - The Portland Group, Inc.

- Des sociétés de services
  - ADINA R&D, Inc.
  - ANSYS, Inc.
  - CPLEX division of ILOG.
  - Fluent Inc.
  - LSTC Corp.
  - MECALOG SARL.
  - Oxford Molecular Group PLC.

1.3 Situer OpenMP parmi d'autre modèle de parallélisation

1.3.1 Communications implicites ou explicites

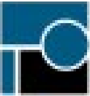
- explicites: Message-passing, MPI, PVM,
- implicites : HPF,OpenMP.



Comparaison de ces modèles (SC 98)

	Message passing	Threads	OpenMP
Portabilité	***	*	**
Conformité au code séquentiel	*		***
Extensibilité/Scalabilité	***	***	***
Performance générale	***	***	***
Support du data-parallélisme	***		***
Parallélisme incrémental			***
Programmation de haut niveau			***
Débogabilité			*** Dichotomie

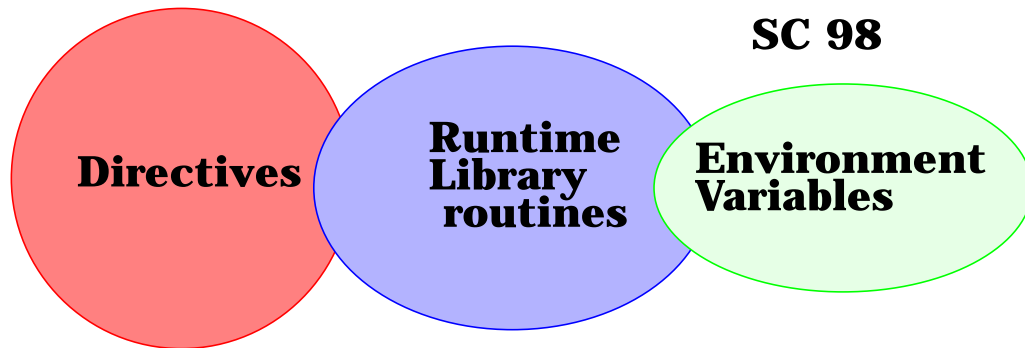
1.3.2 Les architectures adaptées



OpenMP

1.4 Structures d'OpenMP

1.4.1 L'architecture générale



☞ Les directives et leurs clauses sont de 3 types :

- partage du travail (***work-sharing)***,
- partage des données,
- synchronisation.

☞ Les directives ne sont que des commentaires activables par le compilateur grâce une option adéquate. Ainsi, un compilateur ne supportant pas OpenMP les ignorera. C'est ce qui permet de garantir

- **la portabilité des codes OpenMP**,
- la maintenance d'une version **unique**, séquentielle et parallèle, du code.

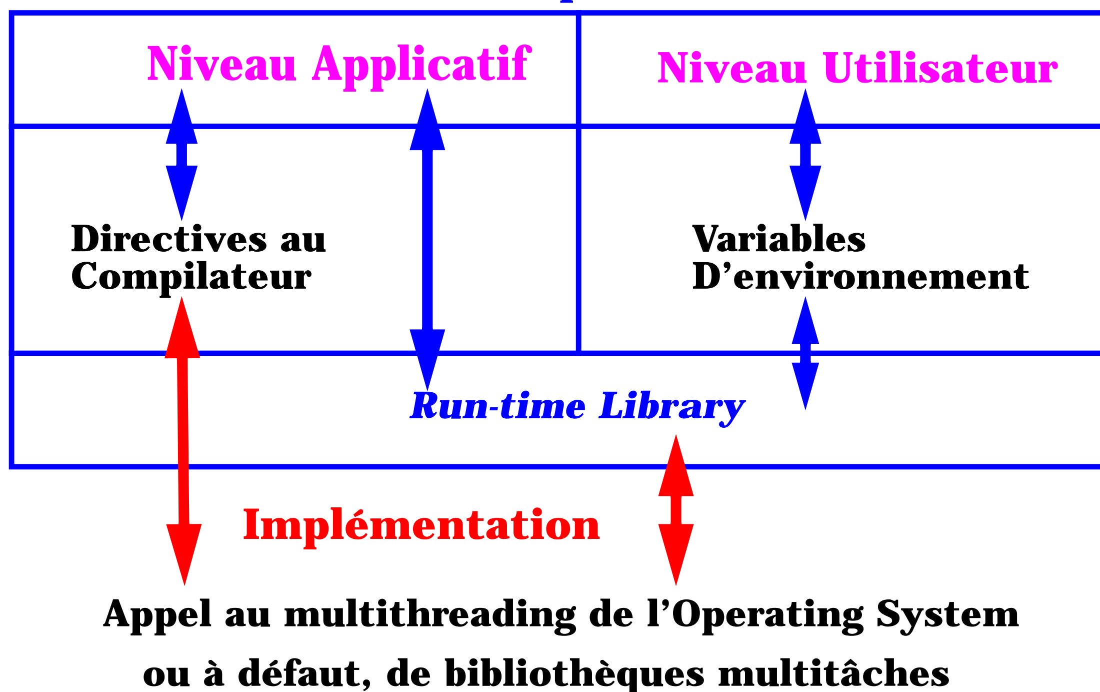
☞ C'est ce que les anglo-saxons nomment la PSE (Portable Sequential Equivalence) au sens faible (Weak PSE). C'est à dire que l'on a cohérence mathématique entre version sé-

quentielle ou parallèle mais pas forcément Strong PSE, c'est à dire équivalence bit à bit des 2 exécutables.

☞ La ***Run-time Library*** et les variables d'environnement permettent de contrôler l'environnement d'exécution comme par exemple le nombre de *threads*

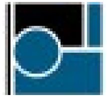
- `call omp_set_num_threads(32)`
- `export OMP_NUM_THREADS = 32`

SC 98 : Issu du tutorial Supercomputing 98



☞ Un certain nombre de produits du marché sont (ou seront) parallélisés avec OpenMP

- bibliothèques tel que NAG SMP, IMSL(?), blas(?), ...
- logiciels commerciaux : Molpro, Nastran, Gaussian2000 bientôt ?
- Les variables d'environnement permettent de garder le contrôle du parallélisme interne de ces "boîtes noires".



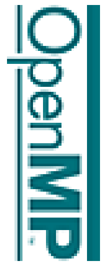
1.4.2 Constructions OpenMP

SC98 : librement adapte du tutorial Supercomputing 98

OpenMP

- ***DIRECTIVES***
  - **Conditional Compilation**
    - **Sentinel !$**
    - **Macro _OPENMP**
  - ***CONTROL CONSTRUCTS***
    - **Parallel Region**
      - **If (...)**
      - **Num_threads() (2)**
    - ***WORKSHARING***
      - **Do**
        - **Schedule ( )**
        - **Ordered**
      - **Sections**
      - **Master**
      - **Workshare (2)**
  - ***DATA CONSTRUCTS***
    - **ThreadPrivate ++**
    - **Data Scope**
      - **Shared( )**
      - **Private( )**
        - **Firstprivate( )**
        - **LastPrivate( )**
      - **Reduction( ) ++**
      - **Copyin( ) ++**
      - **Default( )**
  - ***SYNCHRONISAT. CONSTRUCTS***
    - **Single**
      - **Copyprivate (2)**
    - **Barrier**
    - **Critical**
    - **Atomic**
    - **Ordered**
    - **Flush( )**
- ***CONCEPTS***
  - **Nesting**
  - **Orphaning**
  - **Binding**
  - **Dynamic parallel region**
- ***RUNTIME LIBRARY***
  - **Environment Functions**
  - **Lock Functions ++**
  - **Timers (2)**
- ***ENVIRONMENT VARIABLES***
  - **OMP_SCHEDULE**
    - **Static**
    - **Dynamic, chunk**
    - **Guided, chunk**
  - **OMP_NUM_THREADS**
  - **OMP_DYNAMIC**
  - **OMP_NESTED**

ThreadPrivate ++	Fonctionnalitée ameliorée en version 2
NUM_THREADS() (2)	Nouveauté propre à la version 2



OpenMP

1.5 Terminologie

☞ Il existe un vocabulaire de base propre à OpenMP.

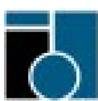
☞ Ils seront traduits, par les termes "francolike" suivants :

- *compliant* : conforme,
- *threads* : tâches plutôt que processus légers,
- *team* : équipe (sous-entendu de tâches),
- *construct* : région, zone ou construction,
- *extent* : littéralement extension, on retiendra le terme de portée[1],
- *statements* : instructions,
- *binding* : relation de parenté,
- *nesting* : nidification (imbrication),
- *orphaning* : orphelinat,
- *region :* région ou zone,
- *scheduling* : ordonnancement,
- *chunk* : paquet ou taille du paquet,
- *granularity* : granularité.
- ....

☞ Le terme implémentation qui existe bien en Français désigne, dans ce cours, la mise en oeuvre d'un compilateur et d'une bibliothèque conforme à OpenMP.

☞ Les terminologies retenues privilégient avant-tout la brièveté.

1. terme dont la traduction est la plus discutable, une traduction en "profondeur" ou "ampleur" aurait été moins ambigue que portée qui est souvent employée pour la programmation séquentielle. La meilleure traduction serait surement champ d'application.



2 - PRINCIPES GENERAUX

2.1 Compilation, chargement et exécution

☞ Sur Cray PVP (J90, SV1), PrgEnv > 3.4 recommandé

```
f90 -x mic toto.f90                # -x mic desactive le microtasking Cray
export NCPUS=4; export OMP_NUM_THREADS=4
ja; a.out; ja -cfst
```

☞ Sur NEC (SX5) : version > 202 recommandée

```
f90 -Popenmp -Wf"-ompctl [no]condcomp" toto.f90        !--- Option de compilation
                                                       !--- et d'edition de liens
export OMP_NUM_THREADS=4; a.out
```

☞ Sur IBM SP3 : version

```
xlf90_r -q smp=omp -q suffix=f=f90
        [[-qsmp=nested_par] [-qsmp=schedule=mode[=chunk]] toto.f90
export OMP_NUM_THREADS=4; a.out
```

ATTENTION

☞ Sur Cray, on peut dynamiquement désactiver OpenMP avec `export NCPUS=1`, l'exécution sera alors séquentielle.

☞ Sur NEC SX5, pour que l'exécution soit séquentielle,

- ☛ il ne faut pas avoir compilé avec `-Popenmp`;
- ☛ l'exécution suivante est considérée comme parallèle.

```
f90 -Popenmp toto.f90;
export OMP_NUM_THREADS=1; a.out
```

OpenMP

2.2 Modèle d'exécution

2.2.1 Le modèle monoprocessus

☞ les tâches (*threads*) sont des "sous-processus" aussi appelées processus légers.

☛La commande `ps` qui permet de voir les processus a souvent une option (`-T` sur NEC ou SGI ou `-o THREAD` sur IBM) permettant de visualiser ces "sous-processus".

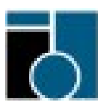
```
NEC-SX5:> a.out &; ps -T
  PID    TID    MID TTY       TIME  COMMAND
10093      0      1 pts/61    0:00   a.out
10093      1      1 pts/61    0:00   a.out
...
```

☛`notion de TID` : numéro de tâche en machine (*TASK ID)*

☞ par défaut, les tâches accèdent à toutes les ressources (mémoires, descripteurs de fichiers, ...) du processus.

☞ Par contre, elles exécutent une charge de travail différentes grâce à une pile (*stack*), pointeur de pile et pointeur d'instructions propres qui leur sont propres.

☞ Ainsi OpenMP correspond plutôt à un parallélisme de contrôle qu'à un parallélisme de données.



2.2.2 La construction mère : la région parallèle

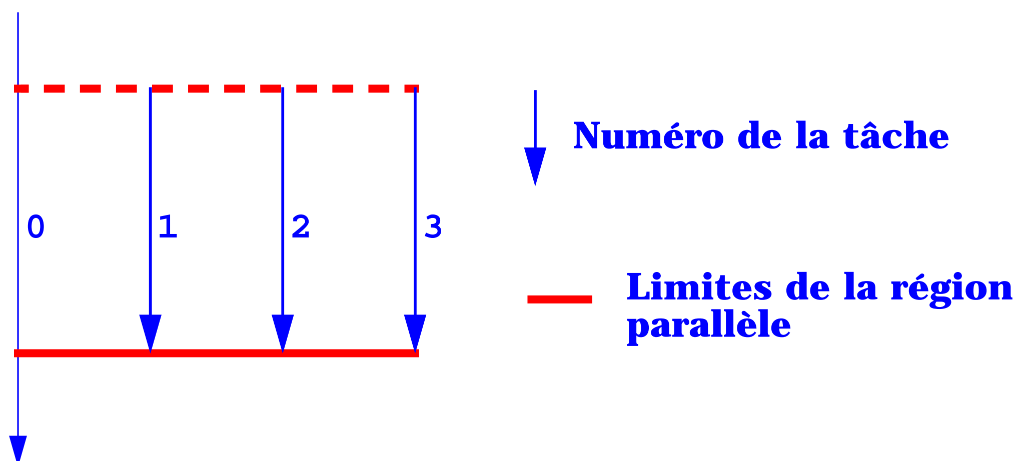
☞ Elles sont basées sur le modèle ***fork and join***.

☞ En d'autres termes, au début de l'exécution,

- le programme est séquentiel,
- seule la tâche maître s'exécute jusqu'à la 1ère région parallèle,
- alors cette tâche crée une équipe de tâches dont elle est la tâche maître.

Exemple A :

```
!$ Initialisation des données
   ...
!$OMP PARALLEL [IF (M > 512)] [clause1 [, clause2] ... ]
   ... région parallèle
!$OMP END PARALLEL
```



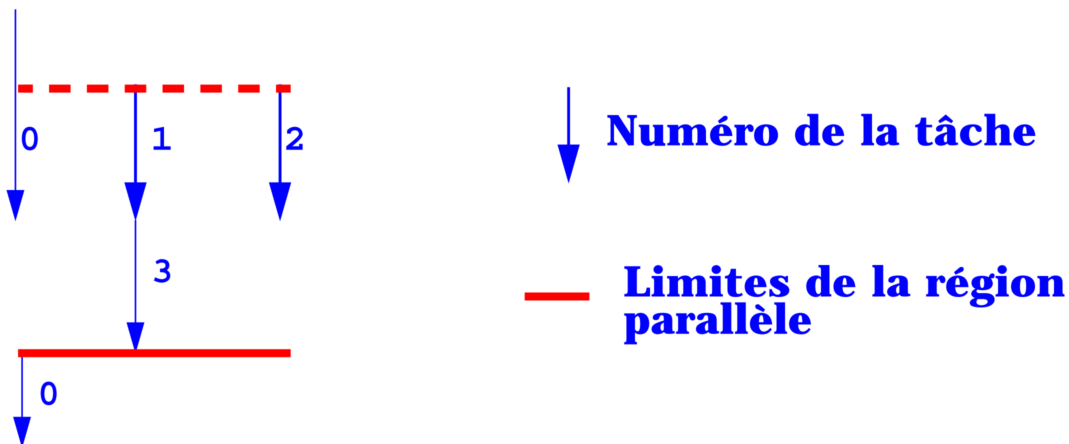
☞ En fin de région parallèle,

- toutes les tâches se synchronisent,
- puis seule la tâche maître poursuit son exécution.
- Les tâches arrivant avant d'autres **dorment** (*sleep*) ou bouclent (***spin waiting***), ce dernier cas peut provoquer une surconsommation CPU.

Attention :

☞ Rien ne garantit le nombre M de processeurs sur lesquels tourneront ces N tâches.

Exemple B : N=4 sur P=3 processeurs



☞ Si les ressources disponibles sont insuffisantes (en mémoire notamment) par rapport aux ressources nécessaires pour créer les tâches demandées, le comportement est dépendant des implémentations

- qui peuvent créer moins de tâches que demandées (NEC-SX5)
- ou faire échouer le programme (IBM NH1).

2.2.3 La numérotation des tâches

☞ Chaque tâche a un numéro dans l'équipe, retourné par la fonction :

```
IAM = OMP_GET_THREAD_NUM()
```

☞ La tâche maître porte le numéro 0.

☞ Le nombre de tâches est déterminé par les variables d'environnement ou par un appel à la ***Run-time Library***.

2.2.4 La portée d'une région parallèle

☞ Les directives de début et de fin d'une région parallèle doivent être dans le même sous-programme.

☞ Les branchements hors d'une région parallèle sont illégaux (i.e. le constructeur doit les repérer et les interdire).

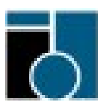
2.2.4.1 La portée lexicale d'une région parallèle

☞ Elle inclut les instructions comprises entre les directives de début et de fin de cette région.

2.2.4.2 La portée dynamique

☞ Elle est constituée par la portée lexicale et les sous-programmes appelés au sein de cette portée lexicale.

☛ voir même les sous-programmes appelées au sein de ces sous-programmes et ainsi de suite.



2.3 Clauses de la directive PARALLEL

☞ La création d'une région parallèle peut être conditionnelle grâce à la clause **IF(expression_logique).**

- ☛cette expression logique sera en fait évaluée avant le début de la région parallèle.

☞ Le nombre de tâches de cette région parallèle peut être fixé grâce à la clause **NUM_THREADS(N > 0) (**)**

- ☛si l'argument de cette clause est une expression scalaire entière, elle sera évaluée avant le début de la région parallèle.
- ☛Clause définie à partir d'OpenMP2 seulement.
- ☛Cette clause a précédence sur le sous-programme **OMP_SET_NUM_THREADS**(N) qui a lui même précédence sur la variable d'environnement **OMP_NUM_THREADS** (voir 7.1 et 7.2)

Les autres clauses possibles sont :

☞ Décrites au chapitre structuration des données

- ☛**PRIVATE**(liste), **FIRSTPRIVATE**(liste), **SHARED**(liste).
- ☛**COPYIN**(liste).
- ☛**DEFAULT(PRIVATE | SHARED | NONE)**.

☞ Décrites au chapitre synchronisations

- ☛**REDUCTION(operator | intrinsic : list)**.

2.4 Les constructions OpenMP *

Définir OpenMP comme un ensemble de directives est très réducteur, mieux vaut le définir comme un ensemble de constructions, elles-mêmes batties sur des directives voir simplement sur des clauses de ces directives.

☞ De façon pratique, une construction se compose en général d'une directive de début et d'une autre de fin pour former une région ou zone d'application.

- ✒Exceptions : constructions **THREADPRIVATE**, **ATOMIC**, **FLUSH**, **BARRIER,**
- ✒Les constructions **REDUCTION, [FIRST|LAST]PRIVATE, SHARED**, ont une fin implicite, celle de la construction dont elles sont définies en tant que clause.

☞ Toutes les constructions OpenMP agisse dans la portée lexicale ou dynamique (concept d'*orphaning*) d'une région parallèle. C'est pourquoi, la construction **PARALLEL/END PARALLEL** peut être percue comme la construction mère.

2.4.1 La portée locale

☞ Certaines constructions ont une portée débutant et s'achevant à leur localisation : **BARRIER, FLUSH**.

☞ D'autres peuvent s'appliquer à l'instruction suivant immédiatement la directive : **ATOMIC**.

2.4.2 La portée lexicale

☞ Elle inclut les instructions comprises entre les directives de début et de fin de cette région.

2.4.3 La portée dynamique

☞ Elle est constituée par la portée lexicale et les sous-programmes appelés au sein de cette portée lexicale.

- ☛ Ainsi que les sous-programmes de niveau 2,3, ...

Exemple C : La construction région parallèle.

```
!$OMP PARALLEL
!--- bloc1
  call sub1()
!--- bloc2
  call sub2()
!--- bloc3
!$OMP END PARALLEL
```

```
subroutine sub2()
  call sub3()
end subroutine sub2()
```

☞ La portée lexicale couvre les blocs 1, 2, 3.

☞ La portée dynamique est constituée de ses 3 blocs mais aussi de sub1, sub2, sub3.

2.5 Règles syntaxiques des directives

Une directive doit être de la forme :

```
sentinelle nom_directive [clause [,clause] ... ]
```

☞ ** Une nouveauté d'OpenMP 2 est d'autoriser les commentaires en fin de ligne en les démarrant par un !

```
sentinelle nom_directive [clause [,clause] ... ]      !--- Comments
```

2.5.1 Sentinelles et directives

☞ Format fixe

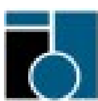
- ✎ la sentinelle recommandée est **C$OMP ;**
- ✎ elle commence forcément en 1ère colonne.

```
C$OMP PARALLEL
```

☞ Format libre

- ✎ la sentinelle est **!$OMP ;**
- ✎ elle peut (précédée de blancs) commencer à partir d'une colonne quelconque.

```
!--- Debut d une region parallele :
     !$OMP PARALLEL
```



☞ Dans les 2 cas les sentinelles, nom de directives et clauses, peuvent être en majuscules ou minuscules.

2.5.2 Directives : lignes suite

☞ Format fixe (caractère de suite en colonne 6)

```
C$OMP PARALLEL DEFAULT(NONE)
C$OMP+ PRIVATE(I,tmp)
C$OMP2 SHARED(A,B,C)
```

☞ Format libre

```
!$OMP PARALLEL DEFAULT(NONE) &
!$OMP PRIVATE(I,tmp)         &
!$OMP SHARED(A,B,C)
```

2.5.3 Compilation conditionnelle par sentinelle ou macro

☞ Une ligne de Fortran ou C peut être compilée conditionnellement de 2 façons :

- ☛ si elle est précédée d'une sentinelle de compilation conditionnelle,
- ☛ grâce à la macro de *préprocessing* **_OPENMP** prédéfinie,
- ☛ ** celle-ci en OpenMP 2 doit être valorisée à un entier de la forme YYYYMM, YYYY et MM étant l'année et le mois de la version supportée par l'implémentation d'OpenMP utilisée

2.5.3.1 Par une sentinelle : !$

☞ Format fixe

☛les sentinelles possibles sont : **!$, C$, *$;**

☛elles commencent forcément en 1ère colonne.

```
C$ Boolean=omp_in_parallel();  print *,'Est-ce une region parallele ? ',Boolean
```

☞ Format libre

☛la sentinelle est forcément : !$;

☛elle peut (précédée de blancs) commencer à partir d'une colonne quelconque.

```
!$  Boolean=omp_in_parallel();  print *,'Est-ce une region parallele ? ',Boolean
```

☞ Un compilateur conforme au standard OpenMP doit alors remplacer cette sentinelle par deux blancs.

2.5.3.2 Par une macro : _OPENMP

☞ Elle doit être définie au niveau des précompilateurs.

☛En OpenMP 2 elle doit être valorisée à AAAAMM=date de la version d'OpenMP à laquelle l'implémentaion est conforme.

☞ Elle doit être automatiquement définie à la compilation d'un code par un compilateur conforme à OpenMP.

```
#IFDEF _OPENMP
  Boolean=omp_in_parallel();
#ENDIF
```

3 - STRUCTURATION DES DONNEES

OpenMP

3.1 Gestion mémoire et processus légers

Les 3 sous-chapitres qui suivent constituent un rappel sur la programmation séquentielle qui pourra être éventuellement contourné. Ils ont pour but d'aider le lecteur peu familier de cette problématique, à caractériser les différents types de variables selon leur zone d'allocation mémoire.

3.1.1 Les différents types de variables.

3.1.1.1 Variables statiques ou automatiques ?

☞ Une variable dont l'emplacement en mémoire est défini dès sa déclaration par le compilateur sera dite **statique**.

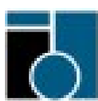
- ☛C'est le cas des variables mises en ***common*** ou ***modules***.

☞ Une variable dont l'emplacement mémoire n'est attribué qu'au lancement de l'unité de programme dans laquelle elle a été déclarée et qui sera désallouée à la fin de l'exécution de celle-ci, sera dite **automatique**.

- ☛La conséquence directe pour une variable automatique est que l'emplacement mémoire qui lui est associé peut varier d'un appel à l'autre.

3.1.1.2 Variables globales ou locales

☞ Une variable est dite globale si elle est déclarée au début du programme principal. Sa durée de vie est alors celle du



programme. Si elle est initialisée à la déclaration[1] elle devient statique sinon elle reste automatique.

☞ Une variable locale est une variable déclarée dans un sous-programme.

☛ Elle a une visibilité[2] réduite à ce sous-programme.

```
subroutine sub()
real :: locale

end subroutine sub
```

☛ Elles se séparent selon les 2 catégories suivantes en fonction de leur durée de vie.

3.1.1.3 Variables locales automatiques

☞ La durée de vie se limite à celle du sous-programme.

☛ Les variables locales sont par défaut automatiques.

```
Call sub()

subroutine sub()
integer, parameter :: m=400, n=500
real :: a
real, dimension(m,n) ::local1
integer, dimension(100,300) :: local2

end subroutine sub
```

☛ `a`,`local1` et `local2` sont des variables locales automatiques.

1. Par un **DATA** par exemple.
2. On préfèrera ici le terme de visibilité à celui de portée pour bien le distinguer d'une part de la notion de durée de vie en **Fortran** et d'autre part de la notion de portée d'une construction OpenMP.

3.1.1.4 Variables locales rémanentes

☞ Il existe 3 cas où les variables locales sont **statiques** :

- initialisation explicite à la déclaration,
- déclaration par une instruction de type **DATA**,
- déclaration avec l'attribut **SAVE**.

```
Call sub()

subroutine sub()
integer, parameter :: m=400, n=500
real, dimension(m,n), save :: local1
data don /0/
integer :: n_fois = 0
...
end subroutine sub
```

- Dans ce cas, elles existent pendant toute la durée du programme en étant allouées à un emplacement mémoire fixe; par conséquent elles conservent la même valeur entre 2 exécutions successives de la même procédure.

Attention : Une variable locale n'est pas nécessairement automatique. En effet, les variables locales désignent souvent, par commodité, les variables locales automatiques alors que des variables locales peuvent très bien être statiques comme les variables locales rémanentes que nous venons de décrire .

3.1.1.5 Les tableaux automatiques

☞ Ce sont des tableaux locaux dont les dimensions dépendent des arguments reçus.

- On parle aussi de tableaux ajustables car leur profil (rang et dimensions) peut changer d'un appel à l'autre.
- Ce sont des variables automatiques dimensionnées également automatiquement.

OpenMP

☛**auto1** et **auto2** sont ici des tableaux automatiques.

```
Call sub(m,n)

subroutine sub(m,n)
integer :: m,n
real, dimension(m,n) ::auto1
real, dimension(100,n) :: auto2
...
end subroutine sub
```

3.1.1.6 Les tableaux dynamiques

☞ Un tableau est dit dynamique si son profil (ensemble de ses dimensions) n'est pas connu à l'écriture du programme mais peut varier d'une exécution à l'autre ou même lors de l'exécution du programme.

☛Ils sont aussi appelés tableaux à profil différé.

☞ Un tableau dynamique voit son emplacement alloué suite à une **demande explicite** du programmeur :

```
integer :: m,n
real, allocatable, dimension(:,:) ::dyn1
m = ...
read *,n

allocate(dyn1(m,n))
```

3.1.2 Les différentes zones mémoires

3.1.2.1 La zone U

☞ C'est la zone système qui sert dès que l'on fait un appel système

- ✒comme par exemple d'Entrées/Sorties (Input/Output).
- ✒C'est une zone de communication entre l'application de l'utilisateur et l'*operating/system* gérant la machine.

3.1.2.2 La zone TXT

☞ C'est celle qui contient le code exécutable obtenu après compilation puis édition de liens.

3.1.2.3 La zone DATA

☞ Zone où sont stockées les données statiques initialisées.

3.1.2.4 La zone BSS

☞ Block Started by Symbol ou Basic Storage Space selon les auteurs : zone où sont stockées les données statiques non initialisées et en particulier les common.

☞ La zone BSS n'est pas stockée dans le fichier exécutable (a.out) sur disque, seule sa taille l'est,

- ✒c'est ce qui explique la différence que l'on peut observer entre un `ls a.out` et un `size a.out`.

3.1.2.5 La pile (*stack*) et le tas (*heap*)

☞ Elles constituent les zones d'allocations mémoires dynamiques dans le sens où leur taille n'est pas connue à priori lors de l'écriture du programme et peut évoluer à l'exécution.

3.1.3 Quelles données pour quelle zone mémoire ?

Zone	Contenu
Zone U	**Kernel UNIX**
TXT	**Code assembleur**
DATA	**Variables statiques initialisées : Data**
BSS	**Variables statiques non initialisées :** ***Common, save***
HEAP	**Tableaux dynamiques : allocate**
STACK	**Variables locales automatiques** **Tableaux automatiques** **Contextes de procédures**

Sur la plupart des plates-formes, on trouve dans :

☞ la zone **DATA**, les variables statiques initialisées,

☞ la zone **BSS**, les variables statiques non initialisées,

☞ le **tas** ou ***heap***, les tableaux dynamiques.

☞ la **pile** ou **stack**, les tableaux automatiques,

OpenMP

3.1.3.1 Cas des variables locales automatiques ?

☞ En **mode *stack***, elles sont allouées dans le *stack*.

- ☛Ce mode impose la valorisation des variables locales, sinon à chaque appel de l'unité de programme leurs valeurs seront différentes.

☞ En **mode statique**, toute variable locale devient permanente en étant allouée dans les segments DATA ou BSS.

- ☛Mode, en général, plus consommateur en espace mémoire.

Attention :

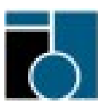
☞ En portant son code d'une plate-forme à une autre, le mode d'allocation des variables locales automatiques peut être différent si on passe d'un compilateur en mode d'allocation statique à un autre en mode d'allocation *stack*.

- ☛ Des erreurs sans incidences jusque là peuvent alors provoquer des résultats différents.

- ☛Elles sont liées à la non initialisation de ses variables locales automatiques qui, en mode *stack,* changeront de valeurs à chaque appel du sous-programme alors qu'en mode statique elles conservaient leur valeur d'un appel à l'autre.

☞ **Il est recommandé de vérifier le comportement de son code avec des compilateurs en mode *stack***.

☞ Comme un compilateur en mode OpenMP sera nécessairement en mode *stack,* aussi considérons-nous dans la suite de ce cours, que toute variable locale automatique est située dans le *stack*.



OpenMP

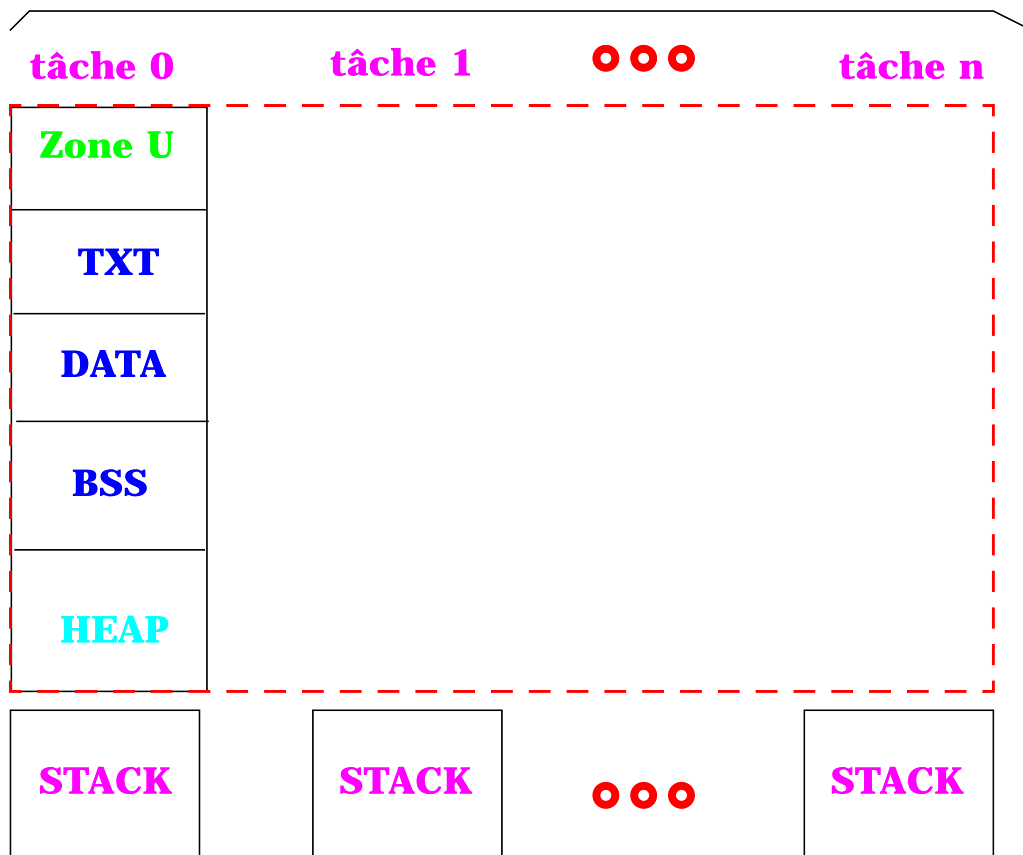
3.1.4 Les tâches OpenMP : des processus légers

☞ A la création d'une région parallèle, chaque tâche ou processus légers ou *threads*) aura sa propre pile (*stack*).

☞ Cependant, chaque tâche continuera d'accéder aux mêmes zones globales qui seront donc dites partagées.

☞ C'est ce que l'on appelle le *Stack Model*

☞ Notons qu'il est indispensable pour un compilateur d'être en **mode *stack*** pour supporter OpenMP.



☞ Les variables **locales** automatiques ou **tableaux automatiques** d'un sous-programme seront donc locaux par défaut à chaque tâche puisqu'alloué dans une zone "propriétaire" à chaque tâche.

Exemple D :

```
!$OMP PARALLEL
  call work(n)
!$OMP END PARALLEL
```

```
subroutine work(m)
  integer :: l,tmp
  real    :: tab_work(m)
    do l= ...
       ...
    enddo
end subroutine work
```

☞ Dans cet exemple, les variables locales **l, tmp** et le tableau automatique **tab_work** ont une occurrence propre sur chaque tâche qu'elles sont seules à pouvoir accéder.

3.2 Attribut des données

3.2.1 Les données partagées : clause SHARED

```
!$OMP PARALLEL SHARED(A)
```

☞ Cette clause définit les variables de sa liste comme partagées par toutes les tâches. C'est à dire visibles par toutes les tâches.

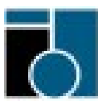
3.2.2 Les données privées : clause PRIVATE

```
!$OMP PARALLEL PRIVATE(x,ytab,i)
```

☞ Une variable privée sera répliquée sur chaque tâche.
- ☛ Son emplacement mémoire n'est plus relié avec celui de la tâche maître.

☞ En entrée de la région parallèle, la valeur d'une variable privée sur chaque tâche sera **indéfinie**
- ☛ même si elle avait été valorisée dans la région séquentielle précédente.

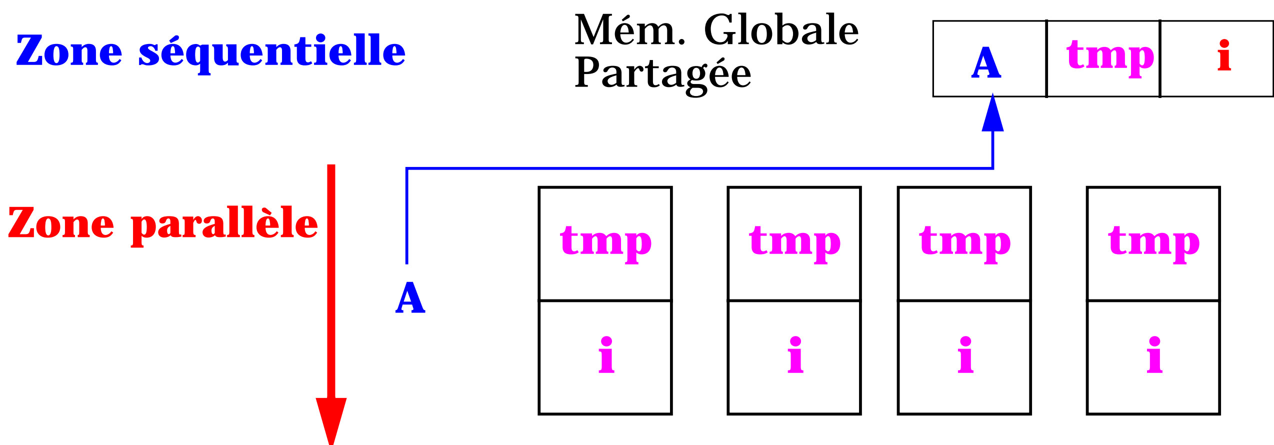
☞ L'attribut privé de cette variable s'applique sur la portée lexicale de la construction OpenMP;
- ☛ par contre, sa durée de vie est celle de la portée dynamique.



☞ En sortie de la région parallèle, la valeur d'une variable privée sur chaque tâche ne sera pas, par défaut, transmise à la région séquentielle qui suit.

Exemple E : SC98 Issu du tutorial Supercomputing 98

```
real, dimension(N) :: A
real               :: tmp

!--- Zone sequentielle
  ...
!--- Zone parallele
  !$OMP PARALLEL PRIVATE(tmp)
    ...
    do i=1,N
    ...
    enddo
    ...
  !$OMP END PARALLEL
```



☞ Les références mémoire sur A sont toutes relatives à la mémoire globale quelle que soit la tâche.

☞ Les indices de boucle dans la portée lexicale de la région parallèle sont tous privatisés par défaut.

3.2.3 Clause FIRSTPRIVATE

☞ La clause **FIRSTPRIVATE** implique la clause PRIVATE.

```
A(:,:) = 1.0
!$OMP PARALLEL FIRSTPRIVATE(A)
 print *,A
!$OMP END PARALLEL
```

☞ Chaque instance privée de *nb* est initialisée avec la dernière valeur de *nb* avant la zone parallèle.

3.2.4 Clause LASTPRIVATE

☞ La clause **LASTPRIVATE** implique la clause **PRIVATE**.

- ☛ Elle n'est valide que pour les directives **DO** et **PARALLEL DO**.
- ☛ Voir directive **PARALLEL DO** au chapitre partage du travail.

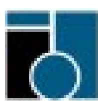
3.2.5 Restrictions sur FIRST et LASTPRIVATE

Ne peuvent être ni LASTPRIVATE, ni FIRSTPRIVATE :

☞ les **pointeurs**, **tableaux dynamiques**,

☞ ***common*** (et leurs **instances**) déclarés en **THREADPRIVATE**,

☞ **tableaux à taille implicite** A(*) **Fortran 77** et **tableaux à profil implicite** A(:) **Fortran 95**.



OpenMP

3.3 La clause DEFAULT( )

☞ Les variables dont la déclaration en **PRIVATE/SHARED** est prohibée, ne sont pas impactées par cette clause.

☞ Par défaut et sans cette clause, une variable est-elle **partagée ou privée** ?

- ☛Les paragraphes suivant répondront à cette question.

3.3.1 DEFAULT(SHARED)

☞ Toutes les variables utilisées dans la portée lexicale de la région parallèle restent partagées par défaut.

☞ Clause commode en *WORK-SHARING* sur des boucles.

3.3.2 DEFAULT(PRIVATE)

☞ Toutes les variables utilisées dans la portée lexicale de la région parallèle deviendront **PRIVATE** par défaut.

☞ Choix judicieux pour une décomposition de domaines !

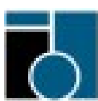
3.3.3 DEFAULT(NONE)

☞ Toutes les variables utilisées dans la portée lexicale de la région parallèle devront être définies avec un attribut

- ☛sinon le compilateur doit protester.

☞ Clause la plus sûre et à conseiller :

- ☛c'est l'***implicit none*** d'OpenMP.



OpenMP

3.4 La directive THREADPRIVATE

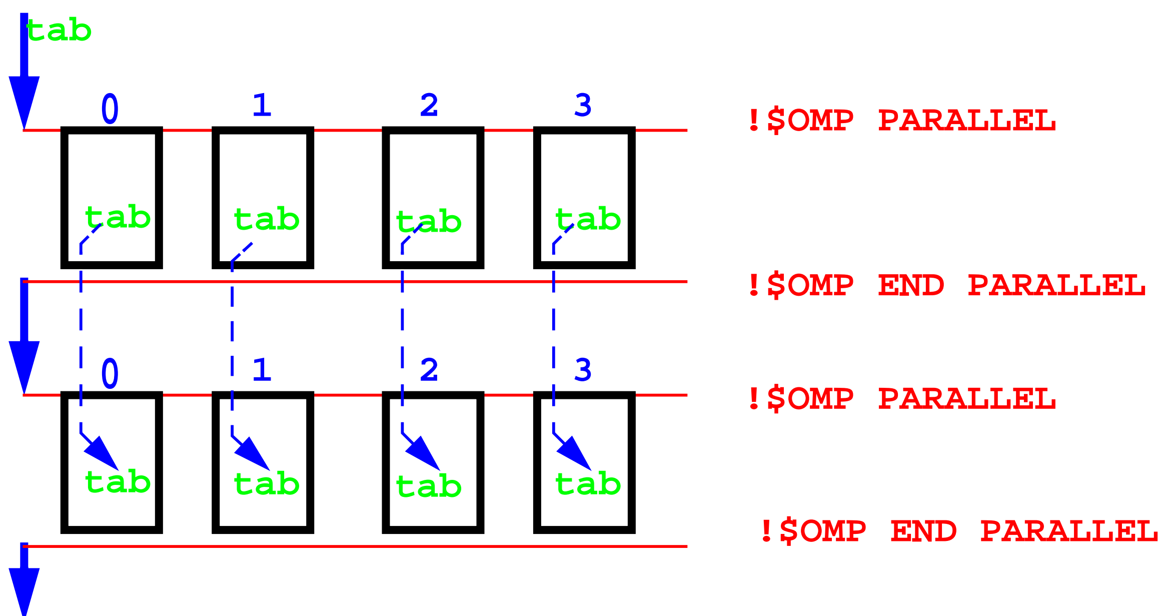
3.4.1 En OpenMP 1

Elle ne peut s'appliquer qu'à des *commons* nommés ou à des instances de ceux-ci.

```
common /commun1/tab
!$OMP THREADPRIVATE (/commun1/)
```

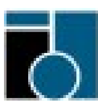
☞ La directive **threadprivate** doit être positionnée

- ☛ après la définition d'un *common* qu'elle inclut,
- ☛ dans chaque sous-programmes utilisant ce *common*.



☞ La portée des variables définies au sein d'un **THREAD-PRIVATE** reste globale pour chaque tâche quelles que soient les régions parallèles, à condition que

- ☛ le nombre de tâches soit constant,
- ☛ le mode dynamique soit inactif (**OMP_DYNAMIC**=FALSE).



ATTENTION : comportement différent du PRIVATE.

☞ Les variables membres d'un **THREADPRIVATE** ne peuvent apparaître dans aucune **clause** exceptée **COPYIN**:

- ☛**PRIVATE**
- ☛**SHARED**
- ☛**FIRSTPRIVATE**
- ☛**REDUCTION** (clause de la directive **DO**)
- ☛**LASTPRIVATE** (clause de la directive **DO**)
- ☛et ne sont pas affectées par la clause **DEFAULT**.

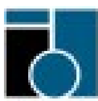
Remarque : si une région séquentielle suit une zone parallèle, une référence à un élément d'un bloc commun déclaré *THREAD-PRIVATE* sera relative à la copie de la tâche maître.

3.4.2 Clause COPYIN des régions parallèles

☞ La clause ne s'applique qu'à un *common* bloc nommé, déclaré en **THREADPRIVATE** ou à une de ses instances.

```
real, dimension(m,n) :: X,Y,Z
common /mem1/X
common /mem2/Y,Z
!$OMP THREADPRIVATE(/mem1/ , /mem2/)

!$OMP PARALLEL COPYIN (/mem1/ , Y, ...)
```

☞ Les données référencées de la tâche maître seront dupliquées au début de la région parallèle dans les instances ***threadprivate*** de chaque tâche.



3.4.3 Exemple de THREADPRIVATE + COPYIN

Exemple F :

```
real, dimension(m,n) :: tab,x
integer              :: moi, omp_get_thread_num
common /commun0/x
common /commun1/tab,moi
!$OMP THREADPRIVATE (/commun1/)

tab(:,:) = 25.0

!$OMP PARALLEL COPYIN(/commun1/)
  moi = omp_get_thread_num()
  if (mod(moi,2)) == 0) tab(:,:) = 10.0
!$OMP END PARALLEL

  tab = 12.0

!$OMP PARALLEL
  moi = omp_get_thread_num()
  print *, 'PE : ', moi,tab(m,n)
!$OMP END PARALLEL
  ...
```

Résultat :

```
export OMP_NUM_THREADS=4
./a.out
PE : 2, 10.0
PE : 1, 25.0
PE : 3, 25.0
PE : 0, 12.0
```

tab = 25.0

effet du COPYIN

0 1 2 3

!$OMP PARALLEL

25.0 if 10.0 tab | 25.0 if 25.0 tab | 25.0 if 10.0 tab | 25.0 if 25.0 tab

!$OMP END PARALLEL

tab=12.0

Région séquentielle

0 1 2 3

!$OMP PARALLEL

tab 12.0 | tab 25.0 | tab 10.0 | tab 25.0

!$OMP END PARALLEL

Double effet du threadprivate

☞ La directive ***THREADPRIVATE*** a un double effet :

- Ces variables deviennent **persistantes**[1] d'une région parallèle à une autre.
- L'instance des variables des zones séquentielles est aussi celle de la tâche 0.

1. A l'image des variables rémanentes en Fortran voir paragraphe 3.1.2.3.

OpenMP

3.5 Les tableaux dynamiques

☞ Ils sont implicitement **partagés** si alloués **hors** de toute région parallèle.

Ils sont privés s'il y a (Au sens OpenMP)

☞ **attribution privée explicite** dans la portée lexicale par la clause **private,**

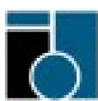
```
real, dimension(:), allocatable :: C

!$OMP PARALLEL PRIVATE(C)
  allocate(C(128) )
  ...                           !--- Code répliquée
  deallocate(C)
!$OMP END PARALLEL
```

☞ **attribution privée implicite** dans un sous-programme appelé au sein d'une région parallèle

```
!$OMP PARALLEL
  call travail(m)               !--- Code répliquée
!$OMP END PARALLEL
```

```
subroutine travail(m)
real, dimension(:), allocatable :: C

  allocate(C (m) )
  ...                           !--- Code répliquée
  deallocate(C)
end subroutine travail
```

ATTENTION : aux implémentations non *thread-safe* et donc non conformes à la norme.



OpenMP

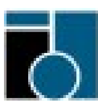
3.6 Les modules Fortran 95 *

Il est nécessaire d'effectuer une mise en garde quant à l'association des modules Fortran 95 dans des sous-programmes appelés au sein de régions parallèles. En effet la zone mémoire de stockage d'un module se situe dans les zones statiques globales à toutes les tâches et non pas dans le *stack*, aussi ne pourra t'on compter sur un typage privé (au sens OpenMP) implicite à la déclaration comme pour une variable locale à un sous-programme. Il faudra donc explicitement privatiser (par la clause PRIVATE) des variables déclarées au sein de modules Fortran 95, mais alors l'association avec le module est perdue.

On peut donc tout simplement considérer qu'avec une implémentation conforme à OpenMP 1.X une bonne stratégie est de ne référencer dans des modules que des variables dont le statut devra être SHARED. Cette limitation sévère pour des codes modulaires écrits avec Fortran 95 saute avec OpenMP 2 qui rend la privatisation explicite de données déclarées dans des modules possible par la directive THREADPRIVATE.

Exemple G : Module, sous-programmes et résultats:

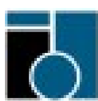
```
module module_omp
  integer, external  :: omp_get_thread_num
  integer, parameter :: m=2,n=2
  real, dimension(m,n) :: local_module
end module module_omp
```



OpenMP

```
program omp_pg
USE module_omp
integer :: moi

!$OMP PARALLEL PRIVATE(moi, local_module)        ! <-- local_module est
                                                 ! privatisée explicitement
!$ moi=omp_get_thread_num()
  local_module = -50 - real(moi)

  !$OMP CRITICAL
  print *,'Thread num : ',moi, ' local_module :', local_module
  !$OMP END CRITICAL
  !$OMP BARRIER
 call travail()

!$OMP END PARALLEL
end program omp_pg
```

```
subroutine travail()
USE module_omp                    !<-- local_module est implicitement SHARED
integer, parameter :: p=2, r=2
real, dimension(p,r) :: local     !<-- local est implicitement PRIVATE
integer :: moi


!$ moi=omp_get_thread_num()

  local_module = -60 - real(moi)
  local        = -70 - real(moi)

  !$OMP CRITICAL
   print *,'Thread num : ', moi, 'local_module : ', local_module,' local : ', local
 !$OMP END CRITICAL

end subroutine travail
```



Résultats :

```
main: 0  local_module :  4*-50.
main: 1  local_module :  4*-51.
sub  : 1  local_module :  4*-60.    local :  4*-71.
sub  : 0  local_module : 4*-60.    local : 4*-70.
```

3.6.1 Extension de THREADPRIVATE et COPYIN **

A partir d'OpenMP 2.0, cette directive et cette clause de la directive PARALLEL peuvent aussi s'appliquer à des variables nommées[1].

☞ Ceci est particulièrement commode dans des sous-programmes appelés au sein de régions parallèles afin de privatiser explicitement :

- Des variables **rémanentes** : **DATA, SAVE**. voir chapitre 3.5.
- Des variables déclarées dans des modules.

Exemple H :

```
subroutine travail()
USE module_omp              !<-- local_module est implicitement SHARED
integer, parameter :: p=2, r=2
real, save, dimension(p,r) :: persiste!<-- persiste est implicitement SHARED

!$OMP THREADPRIVATE( persiste, local_module)


end subroutine travail
```

1. i.e.Le draft OpenMP entend ici par nommée des variables déjà déclarées.

3.7 Statut implicite des variables

Source fréquente d'erreurs notamment en cas de décomposition de domaines.

☞ Dans la portée lexicale d'une région parallèle, les variables non explicitement attribuées (au sens OpenMP) sont implicitement partagées.

☞ Dans la portée dynamique les choses sont plus subtiles:

Attribut implicite des variables d'un ss-prog appelé dans une rég. parallèle

	locales rémanentes	mises en common	déclarées en module	passées en argument	locales	tableaux automatiques
Shared	**X**	**X**	**X**	**X**		
private					**X**	**X**

☛ variables **locales** et tableaux **automatiques** sont implicitement **privées**.

☞ Les variables **locales rémanentes**[1] seront par contre implicitement **partagées**. Il s'agit des variables déclarées

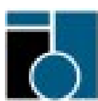
☛avec l'attribut Fortran **SAVE,**

☛ou l'attribut Fortran **DATA,**

☛ou initialisées pendant cette déclaration.

```
integer, parameter :: m=10, n=10

integer, dimension(M,N), save  :: A
real   , dimension(M,N)        :: B, C = 8.0
DATA ((B(i,j), i=1,M), j=1,N) /4.0/
```

1. OpenMP 2 et **Fortran 95** clarifient leur statut en les considérant toutes comme déclarées avec l'attribut **SAVE**.



OpenMP

3.8 Exercice récapitulatif

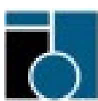
Exemple I : quels sont les attributs (SHARED ou PRIVATE)? Correction en annexe.

```
integer, parameter                  :: p=1024
real, dimension (p)                 :: A, B, C, D
real, allocatable, dimension(:)     :: dyn1, dyn2
common D
common /blank/ B,C
!$OMP THREADPRIVATE (/blank/)
allocate(dyn1(p))

!$OMP PARALLEL
  allocate(dyn2(p))
  do i=1,n
    call work(A,p)
  enddo
!$OMP END PARALLEL
```

Lexical Extent

```
subroutine work(tab,m)
  parameter :: p=256
  real, save, dimension(p) :: t
  real, dimension (p) :: local, B,C
  real, dimension (m) :: autom, tab
  real, allocatable, dimension(:) :: dyn
  integer :: l,tmp
  common /blank/ B,C
  !$OMP THREADPRIVATE (/blank/)

  allocate(dyn(2*m+4))

  ...

  end subroutine work
```

Dynamic extent = Lexical extent + work extent



4 - PARTAGE DU TRAVAIL

4.1 La directive SECTIONS

Exemple J : de parallélisme concurrent

```
!$OMP PARALLEL [clause [, clause ] ]
 ...                                    !--- Mode SPMD
 !$OMP SECTIONS [clause [, clause ] ] !--- Mode MPMD
    !$OMP SECTION
      ...                          !--- block1
    !$OMP SECTION
      ...                          !--- block2
    ...
  !$OMP END SECTIONS [NOWAIT]     !--- Directive obligatoire
   ...
!$OMP END PARALLEL
```

☞ Chaque section n'est plus répliquée sur toutes les tâches mais exécutée par une unique tâche.

☞ Les clauses peuvent être

- ☛**PRIVATE (**List**)**, **FIRSTPRIVATE(**List**)**, **LASTPRIVATE(**List**)**
- ☛**REDUCTION(operator{intrinsic} : list)**.

☞ Méthode non extensible (*scalable*).

Restrictions

Les branchements vers l'extérieur de la portée de cette construction sont prohibés.

Chaque directive SECTION doit être comprise dans la portée lexicale d'une zone SECTIONS - END SECTIONS

4.2 La directive DO

☞ Boucles **privées par défaut** : toutes les tâches exécutent toutes les itérations. Il s'agit de boucle répliquée.

Exemple K :

```
!$OMP PARALLEL
 ...                               !--- Code répliqué
  do i=-4,m,3                      !--- Boucle répliquée
    do j=1,mj                      !--- Boucle répliquée
      A(i) = B(i) + C(i)
    enddo
  enddo
 ...                               !--- Code répliqué
!$OMP END PARALLEL
```

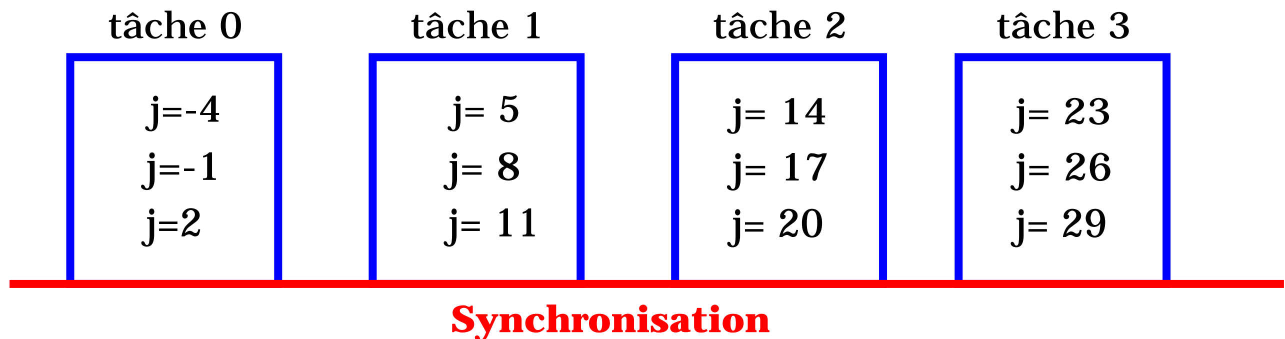
☞ Boucle **partagée** : chaque tâche n'exécute qu'un **sous-ensemble** des itérations de la boucle au sein d'une région parallèle déjà définie.

Exemple L :

```
!$OMP PARALLEL
 ...
 !$OMP DO [clause[,clause] ... ]
   do j=-4,m,3                     !--- Boucle partagée
    do i=1,n                       !-- Boucle répliquée
      A(i,j) = B(i,j) + C(i,j)
    enddo
  enddo
 [!$OMP END DO [NOWAIT] ]          !--- Directive facultative car implicite
 ...
!$OMP END PARALLEL
```

☞ La directive **DO** ne peut s'appliquer qu'à une boucle suivant immédiatement celle-ci.

Exemple M : Pour montrer la répartition des itérations, soit quatre tâches et m= 29. Nous supposerons ici que cette répartition est du type statique (voir le paragraphe 6.3 pour plus de précisions).



☞ Par défaut, il y a synchronisation de toutes les tâches en fin de boucle,

- ☛ sauf si l'on précise la directive **!$OMP DO NOWAIT.**

Les clauses possibles sont :

☞ définies dans ce chapitre

- ☛ **LASTPRIVATE(**list**),**

☞ décrites au chapitre "structuration des données"

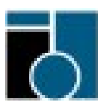
- ☛ **PRIVATE(**list**)**, **FIRSTPRIVATE(**list**)**,

☞ décrites au chapitre "synchronisations"

- ☛ **ORDERED**,
- ☛ **REDUCTION(operator | intrinsic :** list**)**.

☞ décrites au chapitre "concepts"

- ☛ **SCHEDULE(type[,chunk])** .



Restrictions

- Pas sur une instruction **Fortran do while**;
- ni sur une boucle éternelle avec test de sortie;
- l'indice de boucle doit être **entier**;
- les bornes inférieures, supérieures et le pas de la boucle doivent être identiques pour toutes les tâches;
- la directive OpenMP facultative **END DO** doit suivre immédiatement la fin de boucle si l'on désire la préciser;
- une seule clause **SCHEDULE** ou **ORDERED** peut apparaître pour une directive **DO**;
- tout branchement vers l'extérieur de la portée de la directive **DO** est illégal.

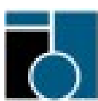
4.2.1 Clause LASTPRIVATE

- Elle ne s'applique qu'aux directives **DO** ou **SECTIONS**.
  - Elle permet de récupérer en dehors de la construction, la valeur de la dernière itération en mode séquentiel.

Exemple N :

```
!$OMP DO LASTPRIVATE(somme,i)
  do i=1,128                              !--- Boucle partagée
    somme = i
  enddo
print *,somme,i
```

```
:>./a.out
128 , 129_ou_128 selon implementation!
```



4.3 La directive WORKSHARE **

Nouveauté de la version 2 d'OpenMP, elle est destinée à permettre la parallélisation d'instructions Fortran 95 intrinsèquement parallèles comme :

- Les **notations tableaux**.
- Certaines **fonctions intrinsèques** (**SUM**, **MATMUL**, ..)
- Le **FORALL**
- Le **WHERE**

Exemple 0 : Issu du *draft* OpenMP 2.0

```
!$OMP PARALLEL PRIVATE(K)

!$OMP WORKSHARE                       !---[ Pas de clause prévues]
   A(:) = B(:)

   !$OMP ATOMIC
   somme = somme + SUM(A)             !--- C'est SUM(A) qui est partagée

   WHERE (A /= 0.0)
     F = 1 / A
   ELSEWHERE (
     F == A
  END WHERE

   FORALL (I=1:M, A(I)/=0.0) F=1/A(I)

   K = SUM(A)                         !--- Attention K est indefini

!$OMP END WORKSHARE  [NOWAIT]

!$OMP END PARALLEL
```

☞ Dans les instructions composées du type **WHERE** et **FORALL**, la partie bloc de contrôle est partagée et exécutée avant les parties d'instructions d'affectation, elles aussi partagées.

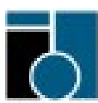
Restrictions

☞ Cette directive ne s'applique qu'à :

- ☛des variables partagées,
- ☛dans la portée lexicale de la construction **WORKSHARE**.

☞ Une fonction ne peut être appelée que si elle est **élémentaire** (***elemental***).

☞ tout branchement vers l'extérieur de la portée, de la directive **WORKSHARE** est illégal.

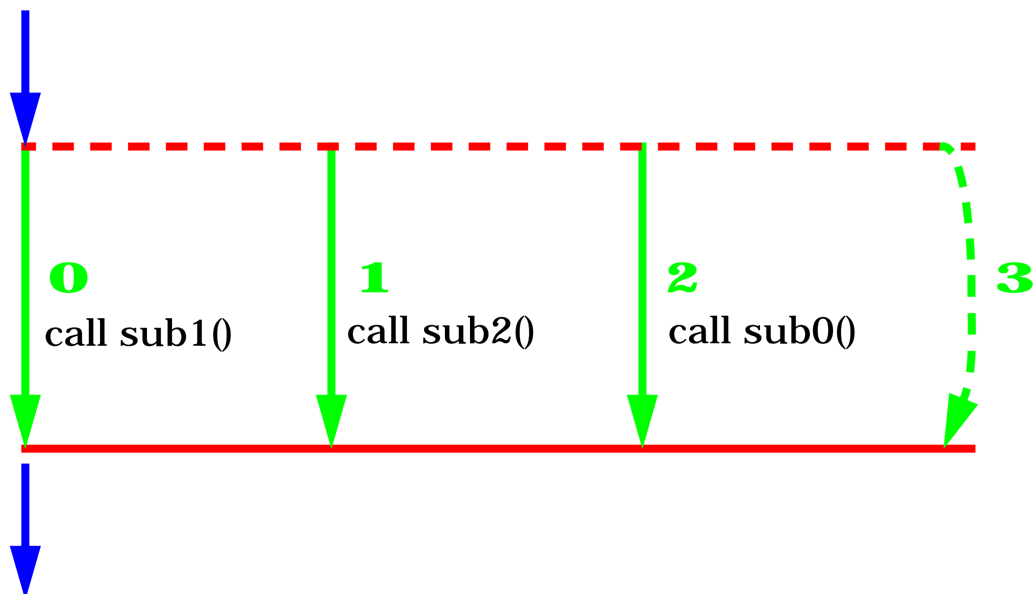


4.4 La directive PARALLEL SECTIONS

☞ Directive condensée pour faire une région parallèle ne contenant qu'une directive SECTIONS.

Exemple P :

```
call omp_set_num_threads(4)
!$OMP PARALLEL SECTIONS
   !$OMP SECTION
   call sub0()
   !$OMP SECTION
   call sub1()
   !$OMP SECTION
   call sub2()
!$OMP END PARALLEL SECTIONS
```



☞ Les clauses acceptées sont l'union de celles des directives PARALLEL et SECTIONS, excepté pour la clause **NOWAIT** (i.e. il y a nécessairement synchronisation en fin de région parallèle).

OpenMP

4.5 La directive PARALLEL DO

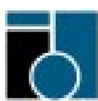
☞ Région parallèle avec une unique boucle partagée.

☞ Les clauses acceptées sont l'union de celles des directives PARALLEL et DO sauf pour NOWAIT.

Exemple Q :

```
!$OMP PARALLEL DO [clause [, clause] ]
  do i=1,m                              !--- Boucle partagee
    do j=1,n                            !--- Boucle repliquee
      do k=1,p                          !--- Boucle repliquee
        A(i,j,k)=k*1000 + j*100 + i*10
      enddo
    enddo
  enddo
[!$OMP END PARALLEL DO ]
```

☞ Seule la boucle suivant immédiatement la directive est partagée.



4.6 La directive PARALLEL WORKSHARE

☞ Région parallèle avec une unique région à partage du travail.

☞ Les clauses acceptées sont l'union de celles des directives **PARALLEL** et **WORKSHARE** sauf pour **NOWAIT**.

Exemple R :

```
integer :: N
real, dimension(N,N) :: A,B,C

NPES = 8
!$OMP PARALLEL WORKSHARE NUM_THREADS(NPES) IF (N**2 > 8*NPES)

  C = MATMUL(A,B)

!$OMP END PARALLEL WORKSHARE
```

☞ La portée de la construction **WORKSHARE** est lexicale.

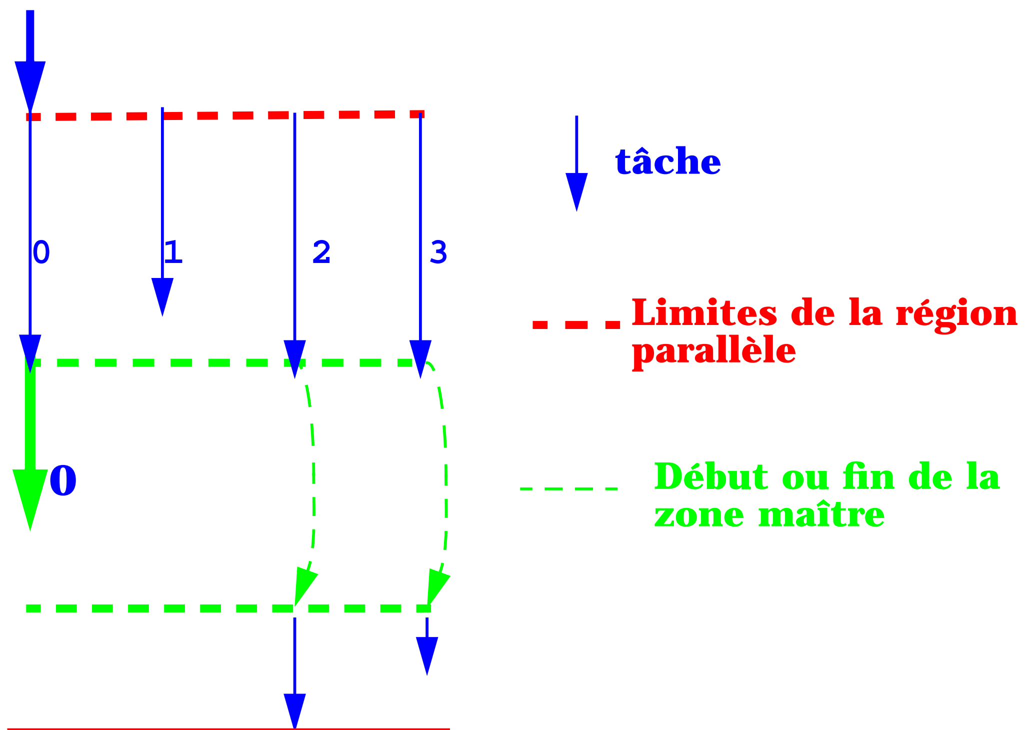
OpenMP

4.7 La directive MASTER

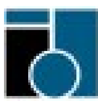
- Une zone maître est exécutée par la seule tâche 0.
- Il n'y a pas de synchronisations en fin de zone maître.

Exemple S :

```
!$OMP PARALLEL
   ...                           !--- Zone repliquee
   !$OMP MASTER
   call sauve()                  !--- Tache 0 uniquement.
   !$OMP END MASTER
   ...                           !--- Zone repliquee
!$OMP END PARALLEL
```



Restrictions : les branchements vers l'extérieur de la zone MASTER sont interdits.



5 - SYNCHRONISATIONS

OpenMP

5.1 Utilité

En ce domaine, OpenMP est assez riche car il réalise la synthèse de ses prédecesseurs.

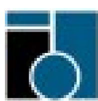
- ☞ !OMP [END] **SINGLE**
- ☞ !OMP **BARRIER**
- ☞ !OMP **ATOMIC**
- ☞ !OMP [END]**CRITICAL**
- ☞ **Directive** et **clause** **ORDERED**
- ☞ **Clause REDUCTION** des **directives DO** ou **SECTIONS**
- ☞ Directive **FLUSH**
- ☞ Verrous (en annexe).

☞ En matière de **performances**, il est préférable de les limiter au strict nécessaire,

- ✒car elles induisent des surcoûts (***overheads***) comme par exemple des temps CPU d'attentes sur des synchronisations (***spin waiting***) non négligeables.

☞ **Débogage** : elles peuvent être très utiles pour ramener le code par dichotomie vers une exécution séquentielle.

☞ Elles peuvent être utile également pour isoler les parties non parallélisables d'une région parallèle.



OpenMP

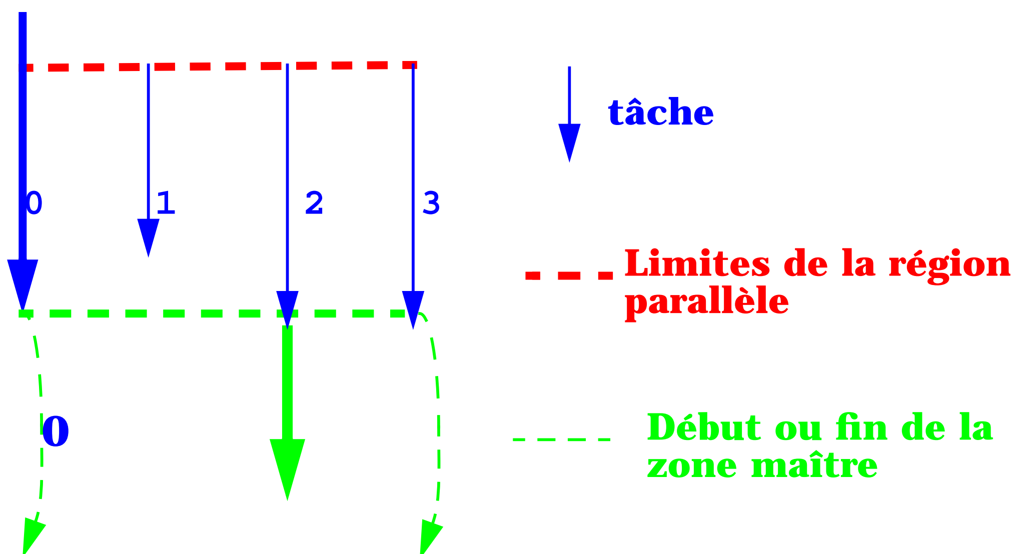
5.2 La directive SINGLE

☞ Une zone **SINGLE** sera exécutée par une des tâches, en général, la première tâche arrivée au début de cette zone.

☞ Les autres tâches attendent celle-ci pour se synchroniser à moins que la clause **NOWAIT** ne soit spécifiée.

Exemple T :

```
!$OMP PARALLEL
  ...                                 !--- Zone répliquée
  !$OMP SINGLE [ PRIVATE(list) | FIRSTPRIVATE(list) ]
  call sauve()
  !$OMP END SINGLE [ NOWAIT | COPYPRIVATE(list)]
  ...                                 !--- Zone répliquée
!$OMP END PARALLEL
```



Restrictions : les branchements vers l'extérieur de la zone SINGLE sont interdits.

OpenMP

5.2.1 La clause COPYPRIVATE **

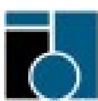
Il s’agit en sortie de la construction SINGLE d’une diffusion (*broadcast*) des valeurs de variables privées vers les instances des autres tâches (i.e. celles qui ne participaient pas à la zone SINGLE).

Exemple U : Diffusion d’une lecture privée

```
!$OMP PARALLEL PRIVATE (donnee, moi)
  moi = omp_get_thread_num()
  !$OMP SINGLE
    read (16) donnee
  !$OMP END SINGLE  COPYPRIVATE (donnee)

  print *,' tache num : ', moi, ' donnee : ',donnee

!$OMP END PARALLEL
```

☞ La clause **COPYPRIVATE** peut s’appliquer à toutes variables de statut privé :

- ✒implicitement (variable locale dans un sous-programme),
- ✒explicitement par la clause **PRIVATE**,
- ✒explicitement par la directive **THREADPRIVATE**.

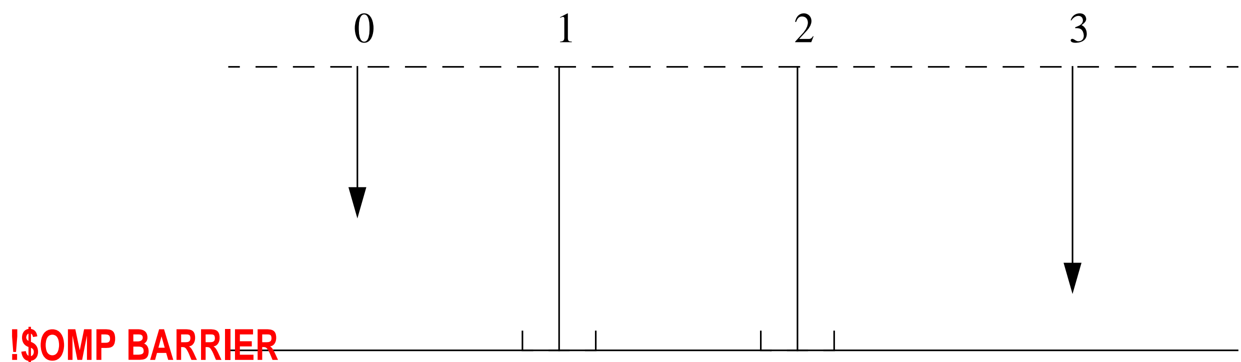


OpenMP

5.3 Les barrières

5.3.1 Les barrières explicites

Positionner une barrière oblige toutes les tâches à attendre que les autres arrivent au même point d'exécution avant de pouvoir toutes continuer.



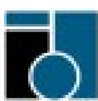
Ici les tâches 1 et 2 sont en attente des tâches 0 et 3.

5.3.2 Les barrières implicites

☞ Elles sont positionnées par défaut en fin de

- ☛région parallèle,
- ☛boucle partagée par la directive !$OMP **DO** ou !$OMP **PARALLEL DO**,
- ☛zone **SINGLE**,
- ☛ zone **SECTIONS**.

☞ On peut inhiber cette barrière implicite par la clause **NOWAIT** en fin de zone excepté pour les constructions **PARALLEL**, **PARALLEL DO** ou **PARALLEL SECTIONS**.

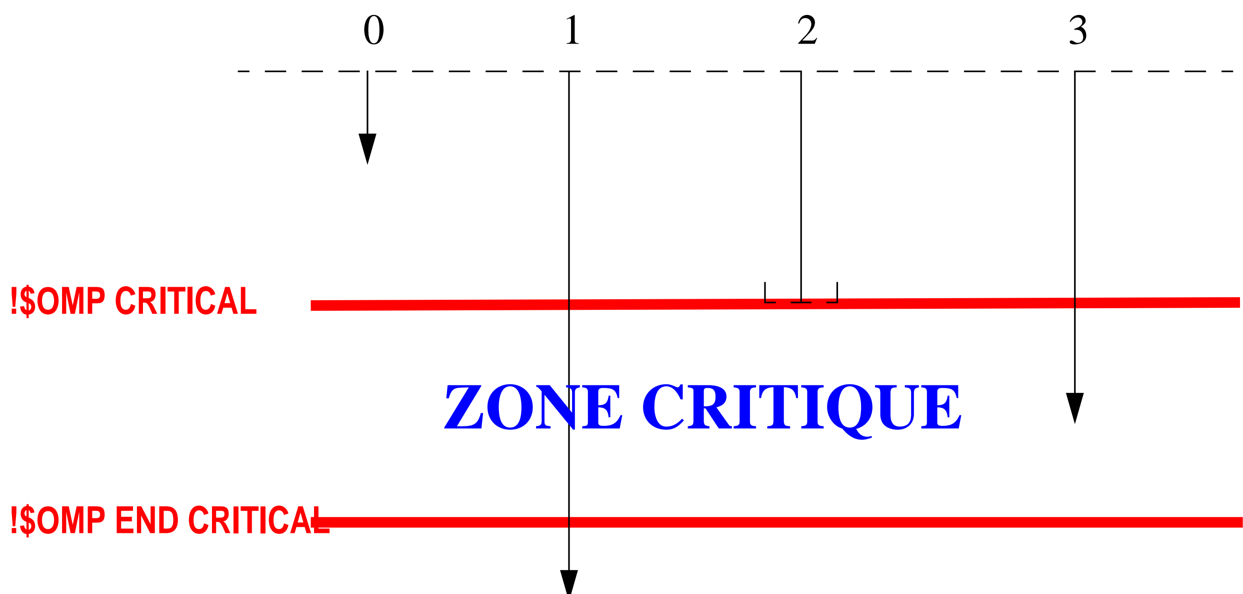


5.4 Les zones critiques : directive CRITICAL

Utilité : définir une zone d'exclusion mutuelle

☞ Portion de code qui ne peut être exécutée que par une tâche à la fois (toutes les tâches finissent par l'exécuter).

```
!$OMP CRITICAL [(nom)]
  ...                        !--- Corps de la zone critique
!$OMP END CRITICAL [(nom)]
```



☞ Une tâche arrivant à la zone critique et autorisée à l'exécuter, en interdit l'accès aux autres tâches tant qu'elle n'en est pas ressortie. Ici, la tâche 2 attend que la tâche 3 soit sortie de la zone critique.

ATTENTION aux noms des zones critiques

☛ **CRITICAL** et **END CRITICAL** doivent avoir le même nom.

☛ 2 zones critiques **disjointes ayant le même nom** constituent une **même zone** critique.

☛ Ainsi, les **zones critiques sans nom** forment une **unique construction** OpenMP.

OpenMP

5.5 La mise à jour atomique : !$OMP ATOMIC

Définition : c'est une mini zone critique sur une unique variable scalaire.

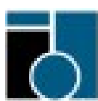
☞ La directive **!$OMP ATOMIC** s'applique seulement sur la ligne **qui suit immédiatement** et qui doit être du type

```
x = x opérateur expr
x = expr opérateur x
x = intrinsic (x , expr)
x = intrinsic (expr , x)
```

- ☛ x étant une variable scalaire de type intrinsèque == > x != tableau, x != type dérivé, etc .
- ☛ expr étant une expression scalaire ne référençant pas x ;
- ☛ intrinsic = [**MAX , MIN , IAND , IOR , IEOR**] ;
- ☛ operator = [**+ , * , - , / , .AND. , .OR. , .EQV. , .NEQV.**] .

☞ C'est une section critique sur l'instruction de lecture et de mise à jour du membre de gauche de l'affectation.

☞ C'est à dire que cette variable x ne peut être lue ou mise à jour que par une tâche à la fois.



Exemple V : Cas de l'adressage indirect

```
!$OMP PARALLEL DO
  do i = 1,n
  !$OMP ATOMIC
    B( A(i) ) = B( A(i)) + C(i)*D(i)
  enddo
```

Exemple W : Cas d'une réduction

```
somme = 0.0
!$OMP PARALLEL DO
  do i = 1,n
  !$OMP ATOMIC
    somme = somme + A(i)
  enddo
```

☞ L'idée sous-jacente est de baser cette construction OpenMP sur une instruction assembleur (**LL/SC**) de lecture et de mise à jour d'une location mémoire avec exclusion de l'accès à celle-ci durant l'opération.

- ☛ C'est ce que l'on entend par atomique pour cette construction OpenMP.

Remarque : l'ordre de passage des tâches dans cette mini-zone critique est imprévisible.

Restrictions : toutes les références à la zone mémoire de x à travers tout le programme doivent avoir le même type. Eviter les équivalences directes ou par le biais des *common*.

5.6 La clause REDUCTION

Elle permet d'effectuer des réductions sur des variables partagées avec les opérateurs associatifs usuels.

- Cette clause peut être précisée pour les **directives**
  - !$OMP **DO** REDUCTION (opérateur : liste)
  - !$OMP **SECTIONS** REDUCTION (opérateur : liste)
  - !$OMP **PARALLEL DO** REDUCTION (opérateur : liste)
  - !$OMP **PARALLEL SECTIONS** REDUCTION (opérateur : liste)

- Les variables de la liste doivent être **SHARED** avant le début de la construction et ne redeviennent définies qu'à la fin de la construction.

- Cette clause s'appliquera aux variables de la liste dans la portée dynamique de la zone parallèle si elles font partie d'instructions du type suivant :

```
x = x opérateur expr
x = expr opérateur x  !--- Ni soustractions, ni divisions
x = intrinsic (x , expr)
x = intrinsic (expr , x)
```

  - x étant une variable scalaire de type intrinsèque == > x != tableau, x != type dérivé, etc .
  - expr étant une expression scalaire ne référençant pas x ;
  - intrinsic = [**MAX , MIN , IAND , IOR , IEOR**] ;
  - operator = [**+ , * , .AND. , .OR. , .EQV. , .NEQV.**] .

☞ C'est une section critique sur les instructions de lecture et de mise à jour du membre de gauche de l'affectation, si elle est citée dans la liste de la clause **REDUCTION**.

☞ C'est à dire que cette variable x ne peut être lue ou mise à jour que par une tâche à la fois.

Exemple X :

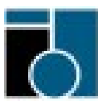
```
somme = 0.0
!$OMP PARALLEL DO REDUCTION (+ : somme)
  do i = 1,n
   somme = somme + A(i)
  enddo
```

Attention :
1 - l'ordre de passage des tâches dans ces mini-zones critiques est imprévisible.
2 - Au sein de la construction, dans l'exemple ci dessus, depuis la directive PARALLEL DO jusqu'au enddo , *somme* ne désigne qu'une instance privée à chaque tâche, ce n'est qu'après la fin de la construction (en l'occurrence fin implicite de boucle) que l'instance partagée de *somme* est mise à jour.

Restrictions : toutes les références à la zone mémoire de x à travers tout le programme doivent avoir le même type. Eviter les équivalences.

☞ En OpenMP 2.0, cette construction pourra s'appliquer à des tableaux[1].

1. Le *draft* est toutefois peu loquace à ce sujet.



OpenMP

5.7 Clause ORDERED de DO et directive ORDERED

☞ Une directive **ORDERED** ne peut être incluse que dans la **portée dynamique**

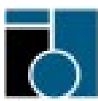
- ✒d'une directive **DO** où **PARALLEL DO**
- ✒affectée de la clause **ORDERED**.

Exemple Y :

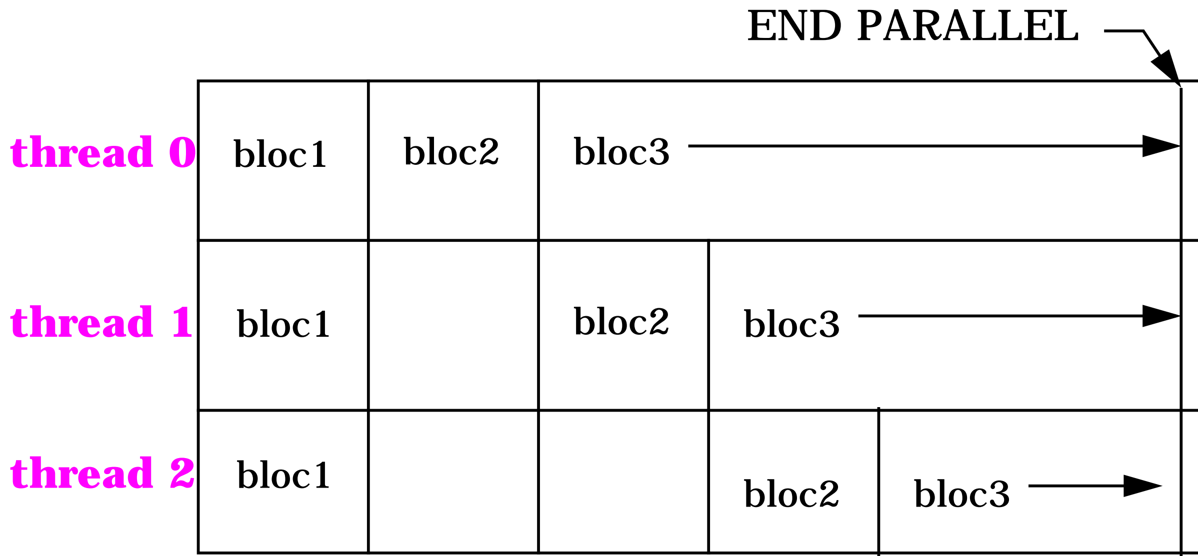
```
!$OMP PARALLEL DO ORDERED
  do i=1,n                             !--- Boucle partagée
    bloc1
    !$OMP ORDERED
    bloc2
    !$OMP END ORDERED
    bloc3
  enddo
[!$OMP END PARALLEL DO]
```

☞ L'idée est que les tâches pénètrent cette zone dans l'ordre séquentiel des itérations.

- ✒En d'autres termes, les tâches exécutent leurs paquets d'itérations respectifs si et seulement si les itérations antérieures de cette boucle sont toutes achevées.



☞SC 98



☞ Une boucle partagée de type ordonnée peut avoir plusieurs zones ordonnées.

☞ Mais une itération ne peut être incluse que dans une unique zone ordonnée. Cc'est un éclaircissement fourni par la version 1.1 .

Restrictions

Les branchements vers l'extérieur de la zone ORDERED sont interdits.

OpenMP

5.8 La directive FLUSH *

Elle permet de définir une vue cohérente de certaines variables entre différentes tâches.

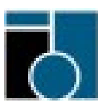
5.8.1 Le vidage implicite de *buffers*

☞ La directive **FLUSH** est en fait implicitement appelée à chaque exécution d'une des directives suivantes :

- ✍ **BARRIER**
- ✍ **CRITICAL / END CRITICAL**
- ✍ **END DO**
- ✍ **END PARALLEL**
- ✍ **END SECTIONS**
- ✍ **END SINGLE**
- ✍ **ORDERED / END ORDERED**

☞ A moins que la clause **NOWAIT** ne soit spécifiée.

- ✍ Ainsi la clause **NOWAIT** comme la directive **BARRIER** a en réalité un double effet, annulant à la fois la synchronisation des tâches et la cohérence de la mémoire.



5.8.2 Le vidage explicite de *buffers*

Si l'on désire mettre en place des synchronisations point à point plus fines, on peut synchroniser les différentes tâches par l'intermédiaire d'une variable partagée. On doit alors "vider" les différents *buffers* ou registres afin de mettre à jour la mémoire physique.

```
!$OMP FLUSH [ (list of variables) ]
```

Exemple Z : schéma producteur - consommateur

Producteur

```
donnee = ...
!$OMP FLUSH(donnee)
drapeau = 1
!$OMP flush(drapeau)
```

Consommateur

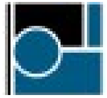
```
drapeau = 0
while (drapeau == 0)
  !$OMP flush(drapeau)
END WHILE
result = f(donnee , ...)
```

☞ La construction ***FLUSH*** assure seulement la cohérence entre la tâche exécutante et les différents niveaux de la hiérarchie mémoire.

☞ D'après la norme, un ***FLUSH*** est nécessaire des 2 côtés pour garantir la cohérence de la mémoire sur n'importe quel type de plate-forme.

Exemple AA : synchronisation avec acquittement

```
  flag = 0; flag2 = 0

!--- On demarre la region parallele
!$OMP PARALLEL PRIVATE(moi,nth)
  moi = omp_get_thread_num();nth = omp_get_num_threads()

  if (moi == 0) then
    donnee = (moi+1) * 1000.0
    !$OMP FLUSH(donnee)
    print *,'tache num : ',moi,' donnee = ' , donnee
!--- Synchronisation en emission
    flag = 1
    !$OMP FLUSH(flag)

!--- Synchronisation en reception pour l'acquittement
    do while (flag2 == 0)
     !$OMP FLUSH(flag2)
    ENDDO
    print *,'tache num : ',moi,' donnee = ' , donnee
  endif

  if (moi == (nth-1)) then
!--- Synchronisation en reception
    do while (flag == 0)
     !$OMP FLUSH(flag)
    ENDDO

    donnee = donnee + (moi+1) * 1000.0
    !$OMP FLUSH(donnee)
    print *,'tache num : ',moi,' donnee = ' , donnee

!--- On envoit un acquittement
    flag2 = 1
    !$OMP FLUSH(flag2)
  endif
!$OMP END PARALLEL
```



Exercice récapitulatif

Determiner l'extention de chaque construction : locales (loc), lexicales (lex) ou dynamiques (dyn)

Questions subsidiaires : 1 - Indiquer (par dyn-n) les constructions dont l'extension dynamique est limitée par du NESTING
2- Quelles constructions ont ici une classification discutable (mettre ! dans les carrés rouges)

- *CONTROL CONSTRUCTS*
  - Parallel Region
    - If (...)
    - Num_threads() (2)
  - *WORKSHARING*
    - Do
      - Schedule ( )
      - Ordered
    - Sections
    - Master
    - Workshare (2)
- *DATA CONSTRUCTS*
  - ThreadPrivate ++
  - Data Scope
    - Shared( )
    - Private( )
      - Firstprivate( )
      - LastPrivate( )
    - Reduction( ) ++
    - Copyin( ) ++
    - Default( )
- *SYNCHRONISAT. CONSTRUCTS*
  - Single
    - Copyprivate (2)
  - Barrier
  - Critical
  - Atomic
  - Ordered
  - Flush

6 - CONCEPTS ET EXEMPLES

OpenMP

6.1 OpenMP et Fortran 95 *

6.1.1 Limitations OpenMP 1 levées avec OpenMP 2

☞ L'interfaçage Fortran-OpenMP 1.X n'a guère été défini. Ainsi les fonctions de la ***Run-time Library*** doivent être explicitement déclarées.

- ☛résolution avec le module **omp_lib (**) qui devra être fourni par l'implémentation.**

☞ Aucune construction n'est prévue pour assurer la parallélisation par OpenMP

- ☛des **notations tableaux (** `A(:) = B(:) + C(:)` **)**,
- ☛des fonctions **intrinsèques** telles que MATMUL.
- ☛Limitation levée par l'introduction de la construction **WORKSHARE** (**)

Néanmoins en OpenMP 1.X :

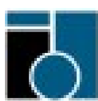
☞ Les **tableaux dynamiques** privés **((de)allocate** F90) ne sont pas prohibés mais doivent être ***thread-safe***.

☞ Toute boucle **Fortran DO** ou **FORALL** dans une région parallèle sera, par défaut, répliquée sur toutes les tâches.

Attention

☞ Aux **variables** définies dans des **modules Fortran 95**, elles ne pourront pas être **privées** dans un **sous-programme** appelé au sein d'une **région parallèle**.

- ☛Limitation levée par l'extension de la directive **THREADPRIVATE** aux données déclarées dans des modules.



OpenMP

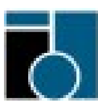
6.1.2 Restrictions explicites en OpenMP 1 et 2

☞ Ne peuvent être déclarés ni **PRIVATE**, ni **FIRSTPRIVATE** ou **LASTPRIVATE** :

- ✒Les **tableaux** à **taille implicite** (**A(*)**).
- ✒Les **tableaux** à **profil implicite** (**A(:)**) grâce à une **interface explicite**

☞ Les **pointeurs** peuvent être privés ou partagés mais ni **FIRSTPRIVATE** ni **LASTPRIVATE**.

☞ Un **pointeur** privé dans une région parallèle sera ou deviendra forcément indéfini à l'entrée de la région parallèle.



6.2 Règles de détermination du statut des variables

Dans une région parallèle, en général,

- Une variable **simplement lue** sera partagée.

- Une variable **simplement écrite** sera,
  - si c'est un **tableau,** partagée,
  - si c'est un scalaire, partagée mais avec mise à jour atomique.

- Une variable **lue avant d'être écrite** peut rester partagée mais mais sous l'effet d'une réduction.

- Une variable scalaire **écrite avant d'être lue** sera privée.

Exemple AB :

```
!$OMP PARALLEL DO PRIVATE(T) REDUCTION(+:somme,err)
DO i=1,N
  DO j=1,M
    T = A(i,j)-B(i,j)
     somme = somme + T
    err = err + T * (A(i,j)+B(i,j))
    C(i,j) = T * poids(i,j)
  ENDDO
ENDDO
```

OpenMP

6.3 Ordonnancement d'une boucle partagée

☞ Il y a 4 **modes** d'ordonnancement s'appliquant uniquement aux directives **DO** ou **PARALLEL DO**

- ☛SCHEDULE(**STATIC**)
- ☛SCHEDULE(**DYNAMIC**)
- ☛SCHEDULE(**GUIDED**)
- ☛SCHEDULE(**RUNTIME**) ---> mode différé à l'exécution par la variable d'environnement : **OMP_SCHEDULE.**

☞ Si le dernier mode est sélectionné, alors le choix est repoussé à l'exécution par la variable d'environnement

```
export OMP_SCHEDULE="mode,chunk"          !--- en ksh
```

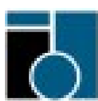
☞ Aucun mode d'ordonnancement par défaut n'est imposé par OpenMP.

6.3.1 Le mode statique

```
!$OMP DO SCHEDULE(STATIC [,chunk])              !--- OU
!$OMP PARALLEL DO SCHEDULE(STATIC [,chunk])
DO i = 1,N
   ...
ENDDO
```

☞ Les itérations sont réparties en paquets d'un nombre fixe X d'itérations successives affectées à une tâche précise

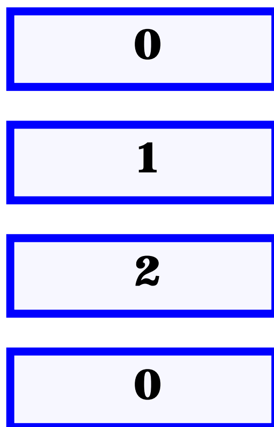
- ☛avec X=***chunk*** s'il est précisé ;

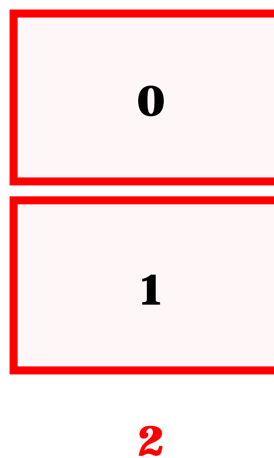


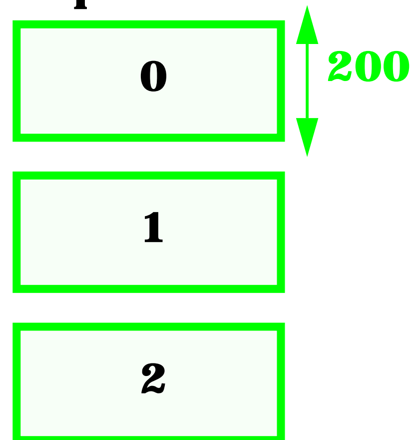
☛X=N/**omp_get_num_threads**, sinon

☞ Les paquets doivent être affectés à des tâches suivant un algorithme de type r*ound-robin.*

Exemple AC : avec N = 600 et 3 tâches







6.3.2 Le mode dynamique

```
!$OMP DO SCHEDULE(DYNAMIC [,chunk])              !--- OU
!$OMP PARALLEL DO SCHEDULE(DYNAMIC [,chunk])
```

☞ Les itérations sont réparties en paquets d'un nombre fixe X d'itérations successives,

☛avec X=***chunk*** s'il est précisée,

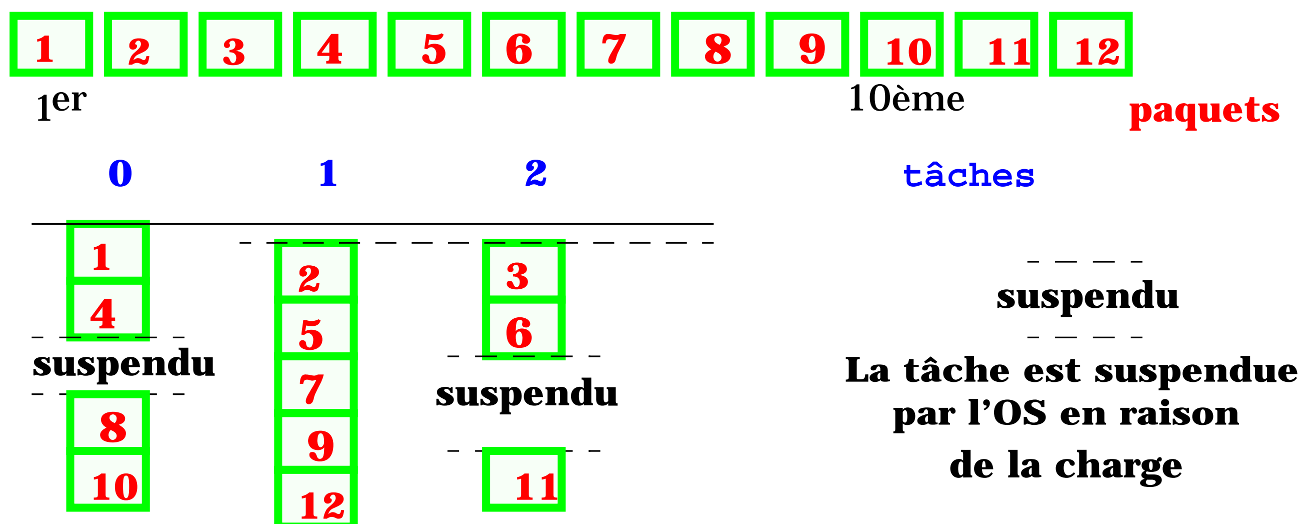
☛X=1,

☛chaque tâche prend le premier paquet disponible.

☞ L'intérêt de ce mode réside dans un éventuel déséquilibre du déroulement des tâches dû à un déséquilibre

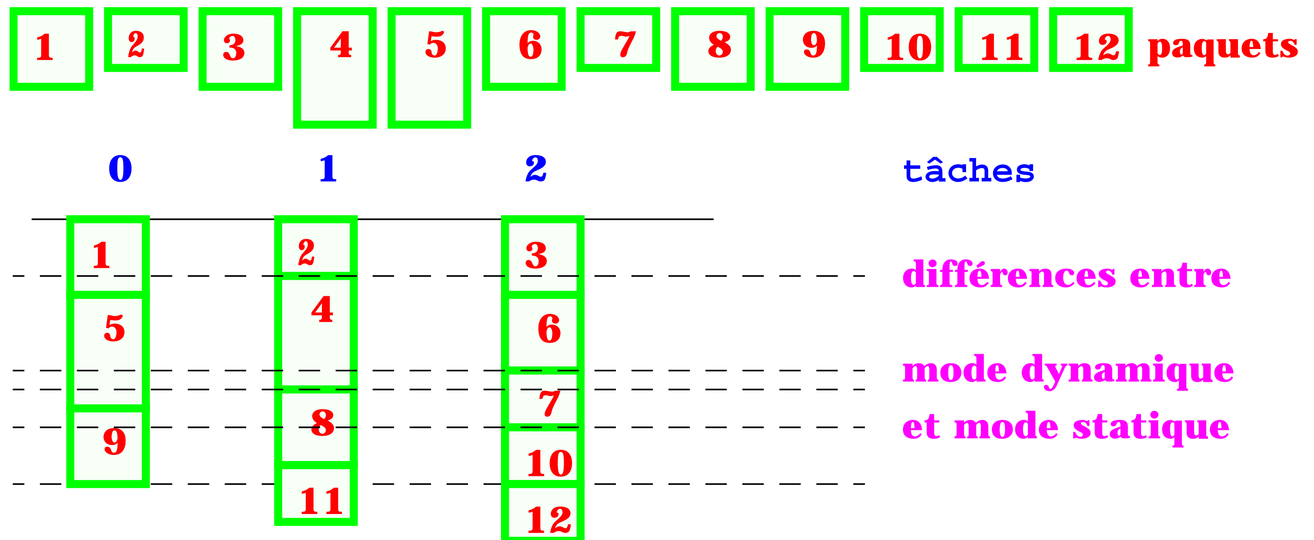
- ☛ de la charge de travail de chaque paquet d'itérations,
- ☛ en raison de la charge du calculateur multi-utilisateurs.

Exemple AD : scénario d'ordonnancement d'une boucle partagée (avec N=600, Ntaches= 3, chunk=50) sur un calculateur en charge



Exemple AE : scénario d'ordonnancement d'une boucle partagée (avec N=600, Ntaches= 3, chunk=50) où les itérations ont une charge variable

```
!$OMP DO SCHEDULE(DYNAMIC [,chunk])          !--- OU
!$OMP PARALLEL DO SCHEDULE(DYNAMIC [,chunk])
DO i = 1,N
   if ( mask(i) <= 0) A(i) = (B(i) - B(i-1)) / 2
    elseif ( mask(i) >= 0) A(i) = sqrt(B(i))
ENDDO
```



6.3.3 Le mode *guided*

```
!$OMP DO SCHEDULE(GUIDED [,chunk])            !--- OU
!$OMP PARALLEL DO SCHEDULE(GUIDED [,chunk])
```

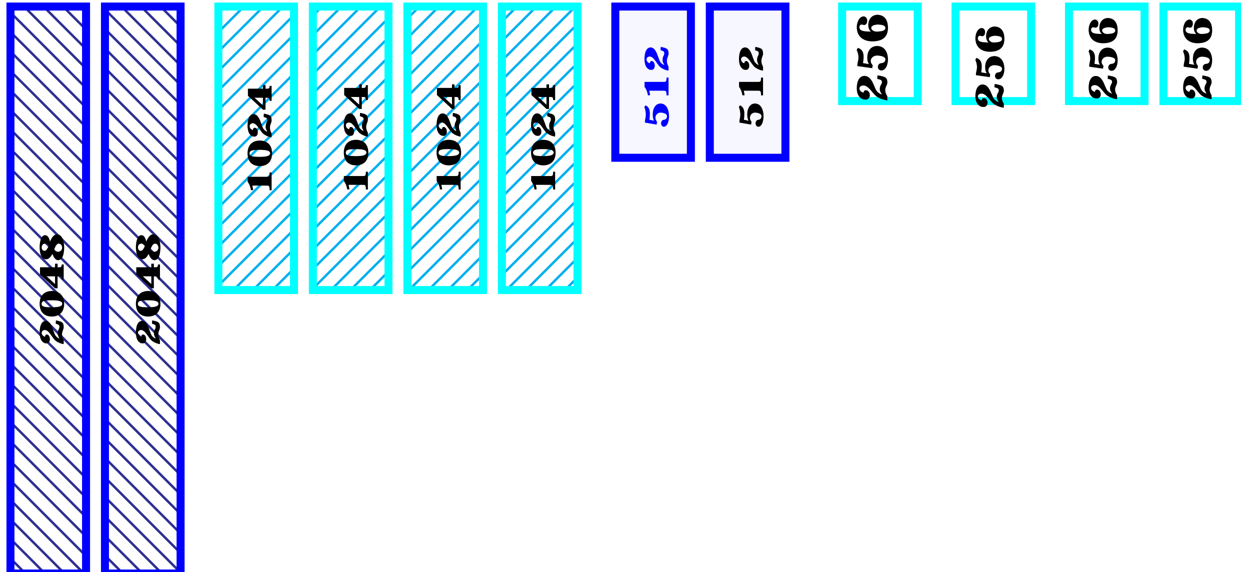
☞ Les itérations sont réparties en paquets de X itérations successives,

- ✒X décroissant exponentiellement au fil des paquets ;
- ✒X ne peut être inférieur à ***chunk*** **excepté pour le dernier paquet** ;
- ✒la taille du paquet initial dépend des implémentations.

☞ Ce mode est intéressant pour un meilleur compromis entre surcoût et équilibrage de charge entre les tâches.

- ✒Plus les paquets sont gros (donc moins nombreux) et moins il y a de surcoûts dûs aux changements de contexte.
- ✒Plus il y aura de paquets de petite taille et plus on a de chances d’équilibrer la charge entre les tâches.

Exemple AF : taille des paquets d'une boucle partagée avec ordonnancement guidé. N=10240, Nchunk = 256



☞ L'idée est donc de maximiser la taille des paquets en début de boucles puis de la diminuer vers la fin de la boucle dans l'espoir d'équilibrer la charge des tâches.

☞ C'est l'opération dite de **pavage** (***tiling***).

☞ L'option *chunk* est particulièrement intéressante sur une machine vectorielle pour éviter de dégrader la longueur de vectorisation,

- ☛ i.e. si l'on parallélise sur la **boucle la plus interne**,

6.3.4 Spécificités du *chunk* indépendant du mode

☞ Le ***chunk*** est toujours une constante pour toutes les tâches

- ☛même si dans le cas du mode ***guided,*** il ne doit pas être confondu avec la taille des paquets qui, elle, peut varier au fil des tâches.

☞ Quel que soit le mode d'ordonnancement des boucles partagées le ***chunk*** doit être :

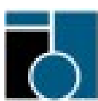
- ☛une expression
- ☛de type entière
- ☛et scalaire.

Exemple AG : Avoir 2 fois plus de paquets que de tâches

```
!$OMP PARALLEL PRIVATE(ntaches)
  ntaches = omp_get_num_threads()
  !$OMP DO SCHEDULE(DYNAMIC, N/(2*ntaches))
  DO I=1,N
    ...
  ENDDO
  !$OMP ENDDO

!$OMP END PARALLEL
```

☞ ***Chunk*** peut être le résultat d'une fonction de type entière et scalaire.

☞ Remarque : le ***chunk*** est toujours évalué avant le début de la construction OpenMP dans laquelle il intervient.



OpenMP

6.4 Régions parallèles dynamiques ou non *

☞ On peut laisser l'implémentation d'OpenMP décider du nombre de tâches de chaque région parallèle en fonction

- ☛ de paramètres propres à l'application,
- ☛ mais aussi de la charge du système.

☞ Le nombre de tâches au sein d'une même région parallèle reste fixe.

☞ On peut le préciser avant la région parallèle par une routine de la ***Run-time Library*** OpenMP

```
!$ call omp_set_dynamic(.TRUE.)
```

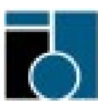
☞ ou par la variable d'environnement :

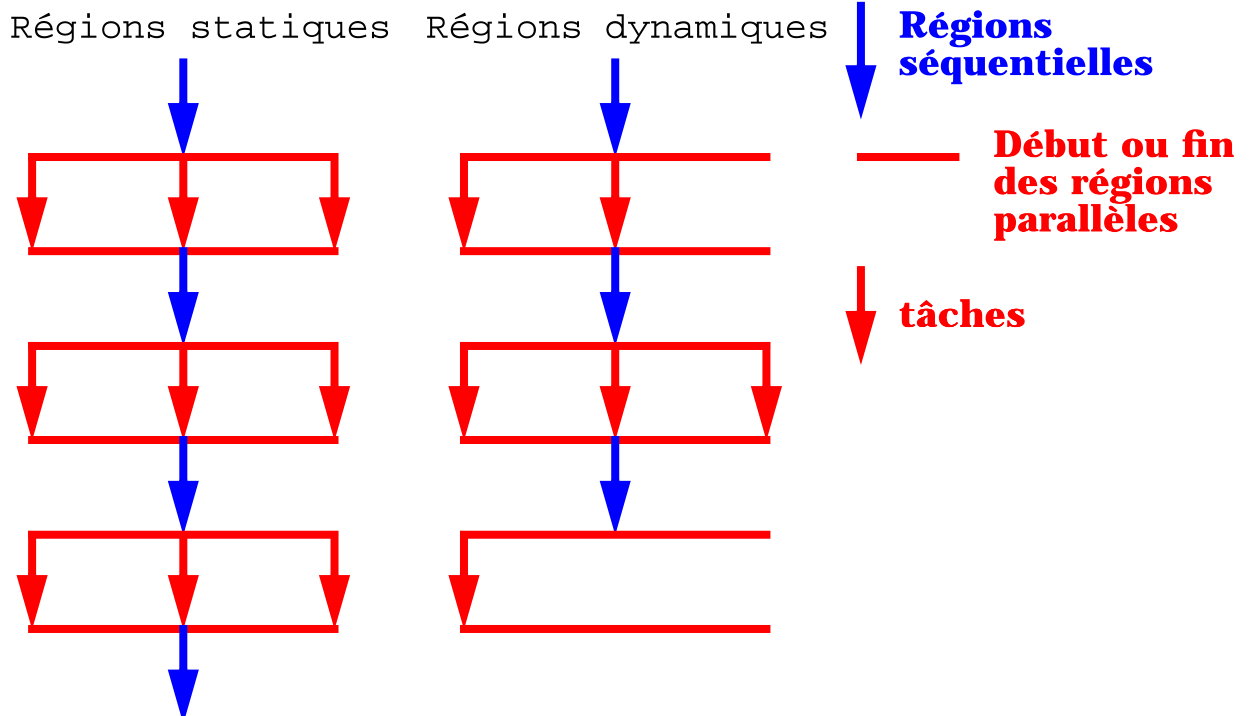
```
export OMP_DYNAMIC=TRUE
```

☞ **OMP_SET_DYNAMIC** l'emporte sur **OMP_DYNAMIC**.

Exemple AH : Positionnement du mode dynamique

```
export OMP_DYNAMIC=TRUE
export OMP_NUM_THREADS=3
a.out
```





ATTENTION

- Le choix par défaut n'est pas imposé par OpenMP.
- Ce choix est parfois le défaut (IBM, ...).
- Si ce mode dynamique est positionné,
  - les contenus de variables attribuées avec la directive ***THREADPRIVATE*** ne sont plus transmis de région parallèle en région parallèle.

6.5 Niveaux de répartition du travail.

Placement dépendant du fournisseur

Ordonnancement au niveau OpenMP

Processeurs

Tâches Physiques en machine (ps -edfT)

Tâches logiques OpenMP

Paquets d'itérations

Itérations

i=1

i=1 à 25

i=976 à 1000

i=1000

Décomposition de domaines

Loop-level parallelism

OpenMP n'étant qu'une API, l'ordonnancement au sein des unités de travail que sont les tâches est seul défini, le "placement" de celles-ci sur les processeurs reste le domaine encore réservé des fournisseurs d'implémentation.

6.6 L'*orphaning*

☞ Dans un sous-programme appelé au sein d'une région parallèle, une directive est dite orpheline (***orphaned***).

☞ Ceci permet d'adopter une programmation **modulaire implicitement** multitâche ou non selon le **contexte**.

☞ Ce sont les directives qui sont orphelines mais par abus de langage on parle aussi de sous-programmes orphelins.

Exemple AI : ***soit 2 unités de programmes distinctes.***

```
!$OMP PARALLEL
  call sub()
!$OMP END PARALLEL
```

```
subroutine sub()
!$OMP CRITICAL
!$ print *,'Si ceci s'affiche, on a compilé en mode OpenMP cette subroutine.
!$  print *,'Mais est on pour autant en mode parallèle ? ', omp_in_parallel()
!$OMP END CRITICAL
end subroutine sub
```

Sur NEC - SX5

```
f90 -c main.o; f90 -c -Popenmp sub.f main.o
export OMP_NUM_THREADS=2
a.out
:>Si ceci s'affiche, on a compilé en mode OpenMP cette subroutine.
:>Mais est on pour autant en mode parallèle ?    F
```

☞ L'appel à **omp_in_parallel** permet de connaître le contexte d'appel de sa propre unitée de programme.

☞ Voir TP1 et 3.

OpenMP

6.7 Le *nesting* *

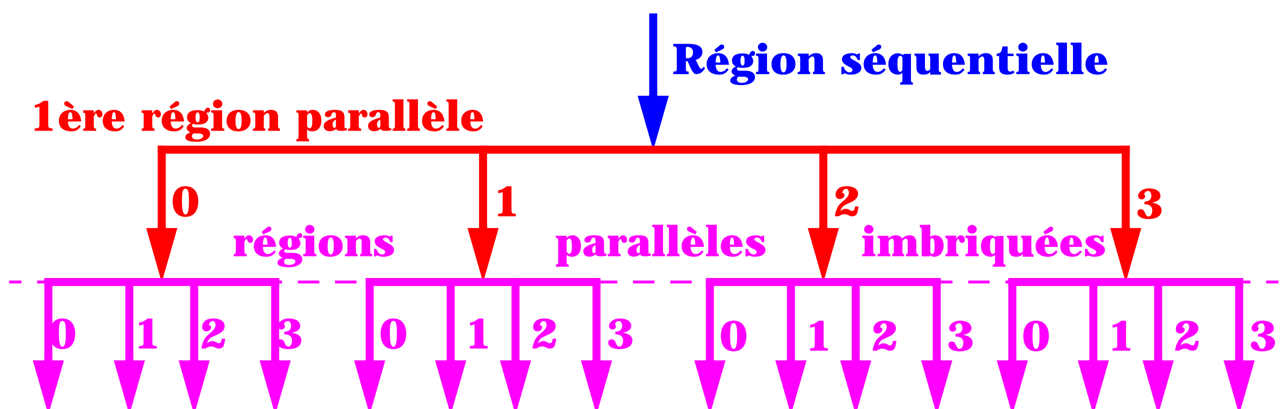
☞ C'est l'**imbrication** de régions parallèles.

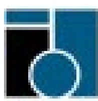
☞ Fonctionnalité rarement implémentée. Par contre, pour être conforme une implémentation devrait au moins permettre la sérialisation des niveaux de parallélisme.

☞ Souvent par une option spécifique
  ☛(**IBM** : `-qsmp=nested_par`)

Exemple AJ :

```
...
!** Fin de la region sequentielle.
!$OMP PARALLEL PRIVATE(moi)
  moi = omp_get_thread_num()
  print *,'reg par 1, tache num : ',moi
  ...
  !$OMP PARALLEL PRIVATE(moi2)
    moi2 = omp_get_thread_num()
    print *,'tache parallele imbriquee num : ',moi,moi2
    ...
  !$OMP END PARALLEL
  ...
!$OMP END PARALLEL
```





6.8 *Binding* *

Il existe des relations de dépendances entre certaines constructions appelées *binding* (relations de parentés).

☞ Art. 1 : Une région parallèle s'apparente à elle même et est éligible pour être apparenté.

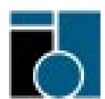
☞ Art. 2 : Les directives **DO, SECTIONS, SINGLE, MASTER,** et **WORKSHARE(**)** s'apparente à la directive **PARALLEL** les encapsulant dynamiquement pour autant que celle-ci existe.

☞ Art. 3 : La directive **ORDERED** s'apparente à la directive **DO** l'encapsulant dynamiquement.

☞ Art 4 : La directive **ATOMIC** force l'accès exclusif en tenant compte des directives **ATOMIC** de toutes les tâches et pas uniquement dans l'équipe courante (--> ***nesting***).

☞ Art. 5 : La directive **CRITICAL** force l'accès exclusif en tenant compte des directives **CRITICAL** de toutes les tâches et pas uniquement dans l'équipe courante (--> ***nesting***).

☞ Art.6 : Une directive ne peut jamais s'apparenter à une quelconque directive qui serait hors de la région définie par la plus proche directive **PARALLEL**.



6.9 Loop-level parallelism

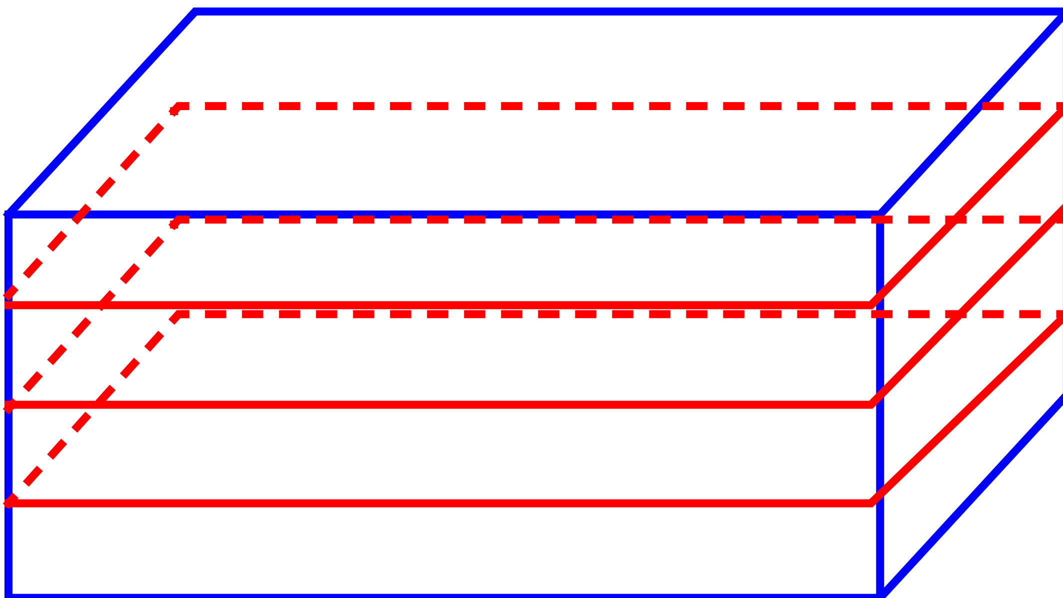
Exemple AK : un classique, la FFT multiple sur un cube

```
call SCFFTM(1, NX, NZ*NY, echelle, X, NX1, Y,     &
                NX/2+1,table,work,0)
```

☞ Ce découpage est-il optimal (mémoire, ...) ?

```
!$OMP PARALLEL DO PRIVATE( X2D, Y2D, WORK)
&
!$OMP SCHEDULE(DYNAMIC,4)
   do j = 1,NZ
     X2D = X(1:NX1,1:NY,j)
     call SCFFTM(1, NX, NY , echelle , X2D, NX1, Y2D,  &
                NX/2+1, table, work,0)

     Y(1:NX/2+1,1:NY,j) = Y2D
   enddo
!$OMP END PARALLEL
```

☞ Réaliser en TP2 le découpage suivant



Exemple AL : Jacobi (sur l'équation de Poisson)

```
!--- Initialisation des coefficients
  ax=1.0/(dx*dx) !--- X-direction coef
  ay=1.0/(dy*dy) !--- Y-direction coef
  b=-2.0/(dx*dx)-2.0/(dy*dy)-alpha ! Central coeff
  error = 10.0 * tol
  k = 1

do while (k <= maxit .and. erreur > tol)
  erreur = 0.0
 !$OMP PARALLEL
 !$OMP DO
   do j=1,m                        !--- Sauvegarde de u
     uold(:,j) = u(:,j)
   enddo
 !$OMP ENDDO

 !$OMP DO PRIVATE(resid) REDUCTION(+:error)
   do j = 2,m-1
     do i = 2,n-1
       !--- Le residu
       resid=(ax*(uold(i-1,j) + uold(i+1,j)) +      &
             (ay*(uold(i,j-1) + uold(i,j+1)) +      &
             b*uold(i,j) - f(i,j))/b
       !--- Solution M.A.J
       u(i,j) = uold(i,j) - alpha * resid
       !--- Cumul des erreurs residuelles.
       error = error + resid*resid
     end do
   enddo
 !$OMP ENDDO NOWAIT
 !$OMP END PARALLEL

 k = k + 1
 enddo                           !--- Fin de la boucle While

print *, 'Nombre d'iterations ', k
print *, 'Residual             ', sqrt(erreur)/dble(n*m)
```

6.10 Décomposition de domaines

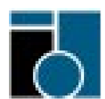
6.10.1 Méthode de décomposition de domaines

☞ La décomposition est manuelle.

Exemple AM :

```
program work
...
real, dimension(M,N,P)::global
...
!$OMP PARALLEL DEFAULT(PRIVATE)         &
!$OMP SHARED(N,somme,Ntaches, global)
  moi = omp_get_thread_num()
  ntaches = omp_get_num_threads()
!--- Calcul des coordonnees et tailles de domaines
  taille = P / ntaches
  ideb = moi * taille +1
  ifin = (moi+1) * taille

  call my_work(ideb, ifin, global)
!$OMP END PARALLEL
  print *,global
end program work
```

```
subroutine my_work(ideb,ifin, global)
integer :: ideb, ifin

do k = ideb , ifin
  do j= 2,n-1
    do i=1,m
     global(i,j,k)=(global(i,j+1,k) -global(i,j-1,k))/2
    enddo
  enddo
enddo
end subroutine my_work
```

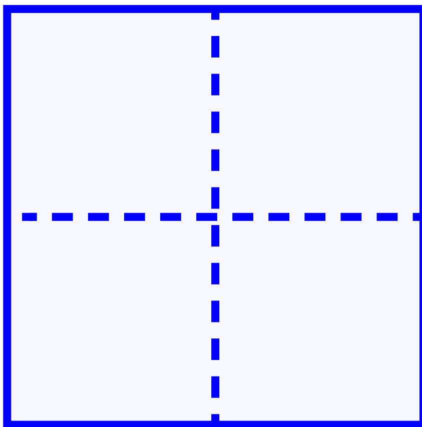
6.10.2 Méthodologie et performances

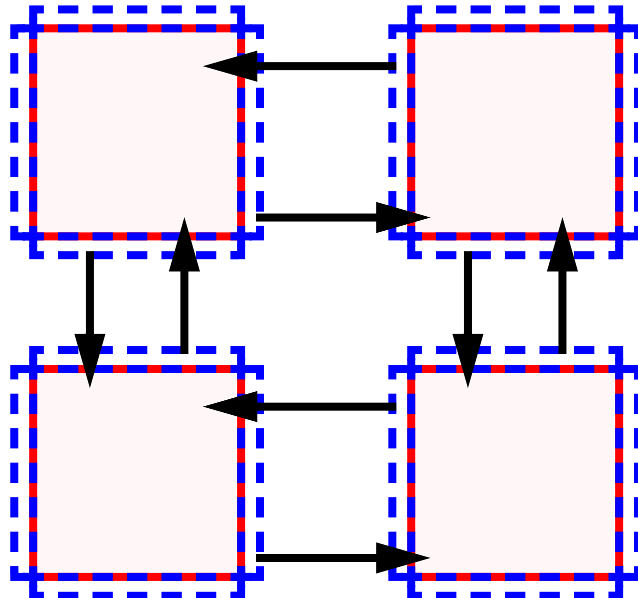
2 options sont alors possibles

- Calculer sur des tableaux partagés : **Simplicité,**
  - nul besoin de cellules fantômes ou de tampons intermédiaires (*ghost-cells* ou *shadow-buffer*).
- Calculer sur des tableaux privés : **performances,**
  - nécessité de synchronisation explicite et de *ghost-cells* ou *shadow-buffer.*

Exemple AN : SC98 partage des données pour Ntâches=4

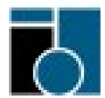
stratégie du "tout partagée" **stratégie du "tout privée"**





Echanges de données **implicites**

Echanges de données **explicites**



OpenMP

6.11 Réflexions sur les performances

Les problèmes d'optimisation avec OpenMP sont très dépendant du type d'architecture mémoire ou de processeurs. Nous ne citerons que les problèmes les plus connus et les plus généraux sans rentrer véritablement dans le détail.

6.11.1 Règles indépendantes des architectures.

☞ En général les constructions **ATOMIC** et **REDUCTION** sont plus restrictives mais plus performantes que la construction **CRITICAL.**

☞ Limiter le nombre de constructions parallèles :

- Eviter d'appeler des régions parallèles dans des boucles ou des sous-programmes appelés plusieurs fois.
- **Toujours** essayer de **paralléliser** la boucle la **plus externe**

☞ Utiliser la clause **SCHEDULE(RUNTIME)** pour pouvoir changer **dynamiquement** en *Loop-level parallelism* l'ordonnancement et la taille des paquets.

☞ La directive **SINGLE** et la clause **NOWAIT** peuvent permettre de diminuer le temps de restitution mais attention aux synchronisations qui doivent alors être explicites.

☞ Il existe un nombre de tâches adapté à la taille du problème or les surcoûts de gestion des constructions

OpenMP s'aggravent avec le nombre de tâches. Inutile d'en demander le nombre maximum dans la plupart des cas.

6.11.2 Risques de dégradation selon la plate-forme utilisée

☞ Chute de la longueur de vectorisation (augmentation du temps USER).

- ☛Eviter **de partager les boucles qui vectorisent (e**n général **les plus internes**[1] sont celles qui vectorisent).
- ☛**Paralléliser les boucles les plus externes**.
- ☛Utiliser la clause `IF(expression_logique)` **si** on a pas d'autres choix que de paralléliser une boucle qui est aussi celle sur laquelle on vectorise.

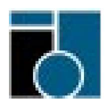
Exemple AO : Garder sur NEC SX5 avec 8 tâches une longueur de vectorisation de 500 minimum.

```
!$OMP PARALLEL DO SCHEDULE(STATIC) IF(N>4000)
do i=1,N
  a(i)=b(i+1)-b(i-1) + 2*b(i)
enddo
!$OMP END PARALLEL
```

☞ Dégradation des accès mémoire : éviter de découper sur la 1ère dimension (**Fortran**) les boucles ou domaines.

- ☛Sur processeur scalaire, mauvaise ré-utilisation des lignes de cache pré-chargées.
- ☛Sur processeur vectoriel, chargement des registres vectoriels moins optimal.

1. Le compilateur peut en effet permuter des boucles imbriquées



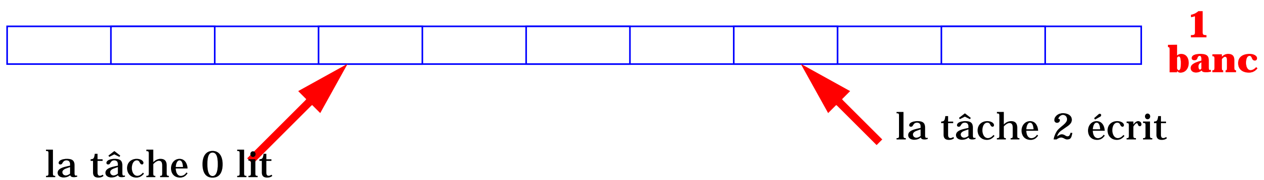
☞ en L*oop-level parallelism,* surcoûts de gestion du parallélisme .

- ☚Ils peuvent être limités au minimum par un bon pavage.
- ☚Ils sont très dépendants de la qualité de l'implémentation.

☞ Conflits mémoire inter-tâches (débit mémoire dégradé).

- ☚Suivant la taille des tableaux, la façon dont ils sont *mapés* (*common* ou non ?), peut provoquer des conflits mémoire sur une machine à mémoire partagée.

Au niveau mémoire
temps de rafraichissemment des bancs mémoire (16 cycles de latence sur NEC SX5)



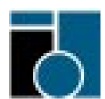
- ☚Se renseigner auprès du constructeur sur les outils disponibles pour diagnostiquer les conflits ou contentions mémoire.

☞ Le placement des données sur mémoire virtuellement partagée.

- ☚Par exemple sur SGI-02000, les données peuvent être placée sur un noeud distant de celui où ont lieu les calculs.
- ☚ Ces accès mémoire de type NUMA[1] sont moins rapides que des accès intra-noeuds.
- ☚OpenMP 2 étant une API[2] de haut niveau, rien n'est pour l'instant prévu pour effectuer un placement de données mieux localisées. Ceci est néanmoins envisagé pour les prochaines versions d'OpenMP.
- ☚Se renseigner auprès du constructeur pour d'éventuelles extensions à OpenMP ou autres solutions propriétaires.

1. Non Uniform Memory Access
2. Application programmeur Interface.

7 - BIBLIOTHEQUES ET VARIABLES



7.1 Les variables d'environnement

- **OMP_NUM_THREADS**
  - Si le mode dynamique est désactivé, elle fixe le nombre de tâches de toutes les régions parallèles.
  - Si le mode dynamique est activé, elle fixe le nombre maximum de tâches que peut comporter l'équipe de chaque région parallèle.

```
export OMP_NUM_THREADS=4                    !--- En ksh
setenv OMP_NUM_THREADS 4                    !--- En csh
```

- **OMP_SCHEDULE**

```
export OMP_SCHEDULE="GUIDED,256"            !--- En ksh
export OMP_SCHEDULE="dynamic"               !--- En ksh
export OMP_SCHEDULE="static"                !--- En ksh
```

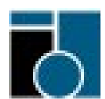
  - Le chunk est optionnel.

- **OMP_DYNAMIC**

```
export OMP_DYNAMIC=FALSE                    !--- En ksh
setenv OMP_DYNAMIC FALSE                    !--- En csh
```

- **OMP_NESTED**

```
export OMP_NESTED=TRUE                      !--- En ksh
setenv OMP_NESTED TRUE                      !--- En csh
```

7.2 La *Run-time Library* d'OpenMP

7.2.1 Généralités

Les procédures ou fonctions de la *Run-time Library* ayant un équivalent parmi les variables d'environnement sont prioritaires localement.

☞ Les ***_SET_*** sont des **sous-programmes** à appeler en général dans les régions **séquentielles**.

☞ Les ***_GET_*** sont des INTEGER ou LOGICAL **functions** à appeler en général dans les régions **parallèles.**

ATTENTION :

☞ En OpenMP 1, aucun interfaçage n'étant défini par rapport à **Fortran 95**, ces fonctions doivent être explicitement déclarées.

☞ Par contre, à partir d' OpenMP 2, les fournisseurs d'implémentations doivent fournir un module **omp_lib** contenant les déclarations de toutes les procédures ou fonctions de la ***Run-time Library*** ainsi qu'une documentation de l'implémentation.

7.2.2 Les sous-programmes de *timing* **

7.2.2.1 Le *timer* standard OpenMP 2

☞ Temps en secondes depuis un point arbitraire dans le passé ?

```
double precision function OMP_GET_WTIME()
```

☞ Il s'agit d'un temps de restitution

- (*elapsed time* ou *wall-clock time*) et non d'un temps CPU,
- **propre à chaque tâche**.

Exemple AP : de mesure du temps

```
double precision debut, fin

!$OMP PARALLEL PRIVATE(debut,fin)
debut = omp_get_wtime()
   ... travail à mesurer
fin = omp_get_wtime()
print *,'temps elapsed : ', fin - debut
!$OMP END PARALLEL
...
```

Attention : selon l'ordre d'exécution des tâches et la charge de la machine, la mesure de temps peut varier d'une tâche à l'autre et d'une exécution à la suivante.

7.2.2.2 Précision du *timing* OpenMP

```
double precision function OMP_GET_WTICK()
```

- Précision données en secondes.

7.2.3 Les sous-programmes relatifs au contexte

7.2.3.1 Fixer ou connaître le nombre de *threads*

```
subroutine OMP_SET_NUM_THREADS(N)
integer N                          !--- serial region only
end                                !--- undefined in // region
```

☛ En mode dynamique, le nombre de tâches fixées est celui de la seule zone parallèle suivante

```
integer function OMP_GET_NUM_THREADS()
OMP_GET_NUM_THREADS = 1            !--- parallel region only
end
```

☛ Dans une région parallèle imbriquée, c'est le nombre de tâches de cette région parallèle imbriquée qui est retourné.

Attention : au sein d'une région parallèle imbriquée mais sérialisée, cette fonction retourne 1 et non le nombre de tâches de la région parallèle supérieure.

7.2.3.2 Connaître mon numéro de tâches

☞ Indispensable en décomposition de domaines

```
integer function OMP_GET_THREAD_NUM()
OMP_GET_THREAD_NUM = 0                      !--- everywhere
end
```

Attention : au sein d'une région parallèle mais imbriquée et sérialisée, cette fonction retourne 0 et non le numéro de *tâche* de la région parallèle supérieure.

7.2.3.3 "To be or not to be in // region !"

☞ Notamment intéressant dans un sous-programme et en décomposition de domaines.

```
logical function OMP_IN_PARALLEL()
OMP_IN_PARALLEL = .FALSE.                !--- everywhere
end
```

Attention : au sein d'une région parallèle imbriquée mais sérialisée, cette fonction retourne .FALSE. .

7.2.3.4 Nombre maximum de *threads*

```
integer function OMP_GET_MAX_THREADS()
OMP_GET_MAX_THREADS = 1                  !--- everywhere
end
```

☛la valeur de cette fonction est affectée par un éventuel appel à **OMP_SET_NUM_THREADS**.

7.2.3.5 Nombre de processeurs

```
integer function OMP_GET_NUM_PROCS()
OMP_GET_NUM_PROCS = 1                !--- everywhere
end
```

7.2.3.6 Le mode dynamique des régions parallèles

☞ Veut-on que les régions parallèles aient toutes le même nombre de tâches ou laisse-t-on l'implémentation choisir ?

```
subroutine OMP_SET_DYNAMIC(Flag)
logical flag                                  !--- everywhere
end
```

```
logical function OMP_GET_DYNAMIC()
OMP_GET_DYNAMIC = .FALSE.                     !--- everywhere
end
```

☛ Une implémentation peut ignorer le mode dynamique.

☛ Une implémentation peut aussi être par défaut dynamique.

7.2.3.7 *Nesting or not nesting* ?

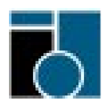
```
subroutine OMP_SET_NESTED(Flag)
logical flag                                  !--- everywhere
end
```

```
logical function OMP_GET_NESTED()
OMP_GET_NESTED = .FALSE.                      !--- everywhere
end
```

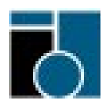
☛ L' imbrication de boucles parallèles doit être supportée.

☛ Mais l'implémentation garde toute latitude sur le nombre de *threads* d'une région parallèle imbriquée.

☛ En conséquence, l'imbrication de boucles parallèles peut être sérialisée.



8 - CONCLUSIONS



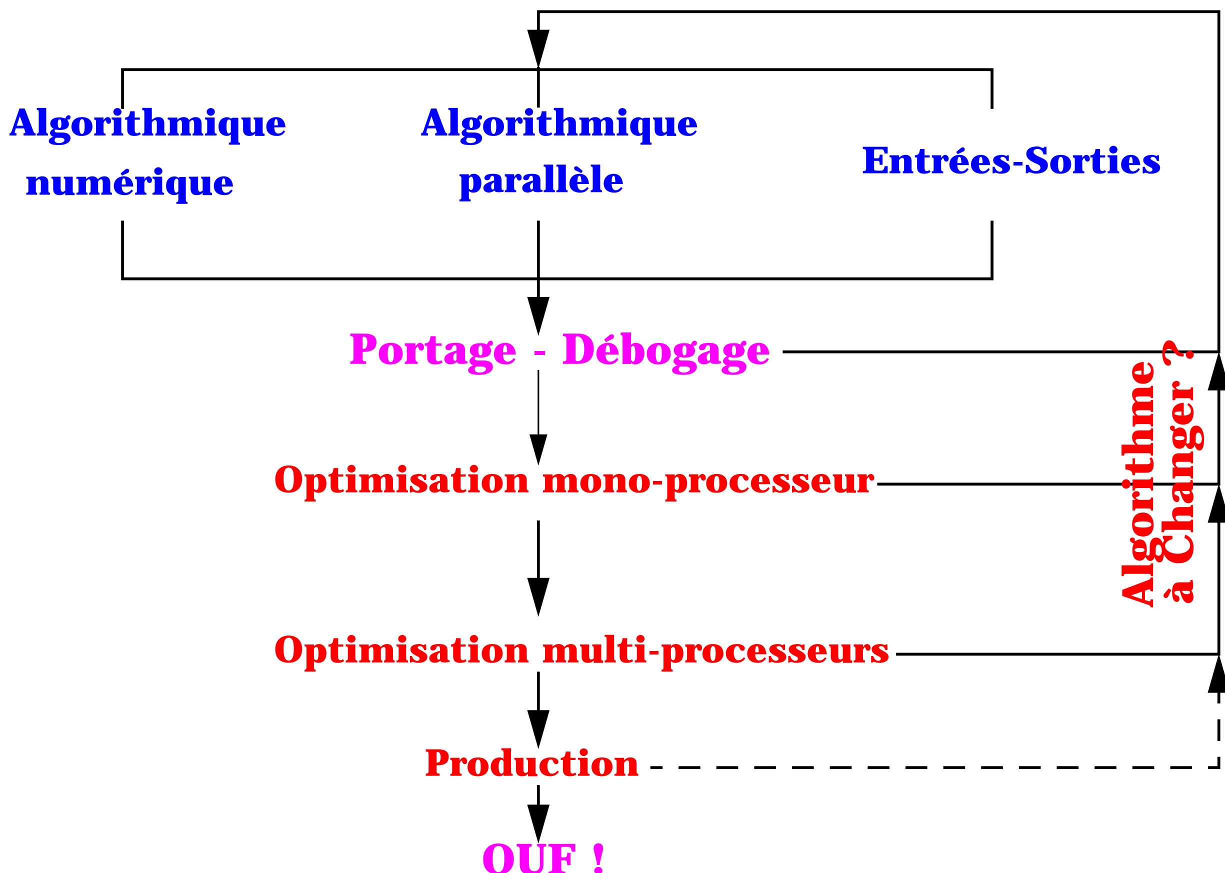
8.1 Méthodologie d' "OpenMPisation"

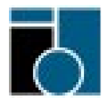
2 grandes méthodes sont possibles

- Approche intégrée plutôt pour les applications **futures**
  - approche à **gros grain** (modèle **SPMD**) ;
  - en profiter pour (re)mettre à plat la structure des données.

- Parallélisation possible et rapide de codes **pré-existants**
  - approche à **grain fin** (***Loop-level*** *parallelism*) ;
  - peut être assistée par des bibliothèques ***Openmpisé*** ou par la parallélisation **automatique** (***autotasking***).

8.1.1 Approche intégrée





8.1.2 Approche progressive

MÉTHODE | **OUTILS**

☞ Rares sont les plates-formes à offrir tous ces outils, consultez les documentations, publicités, renseignez-vous auprès des constructeurs, supports utilisateurs, voir également au paragraphe 8.

OpenMP

8.2 Les atouts d'OpenMP

☞ **Réduire** les **temps de restitution** d'une application **sans en changer l'algorithmique numérique.**

- ☛quand les temps de calculs sont prohibitifs.
- ☛quand cette application prend beaucoup de mémoire et qu'elle n'est pas parallèle. la plupart des processeurs restant alors inactifs (***idle***).

☞ Avantages d'une parallélisation par OpenMP

- ☛**lisibilité**, **évolutivité**,
- ☛**débogage**,
- ☛**unification** des **versions parallèles** et **séquentielles (PSE)**.

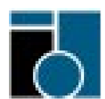
☞ Avoir conscience des pièges et concepts est un atout décisif pour

- ☛éviter les *bugs* classiques,
- ☛l'optimisation des performances.

ATTENTION : même avec un modèle de communication implicite, il est à la charge du programmeur d'éviter tous les problèmes classiques du parallélisme :

- ☞ **Verrous mortels** (***deadlock***).
- ☞ **Boucles éternelles** (***livelocks***).
- ☞ **Conflits** sur des variables.
- ☞ **Effets de courses** (***race conditions***).

pouvant rendre les résultats incorrects même sans messages à la compilation ou à l'exécution.

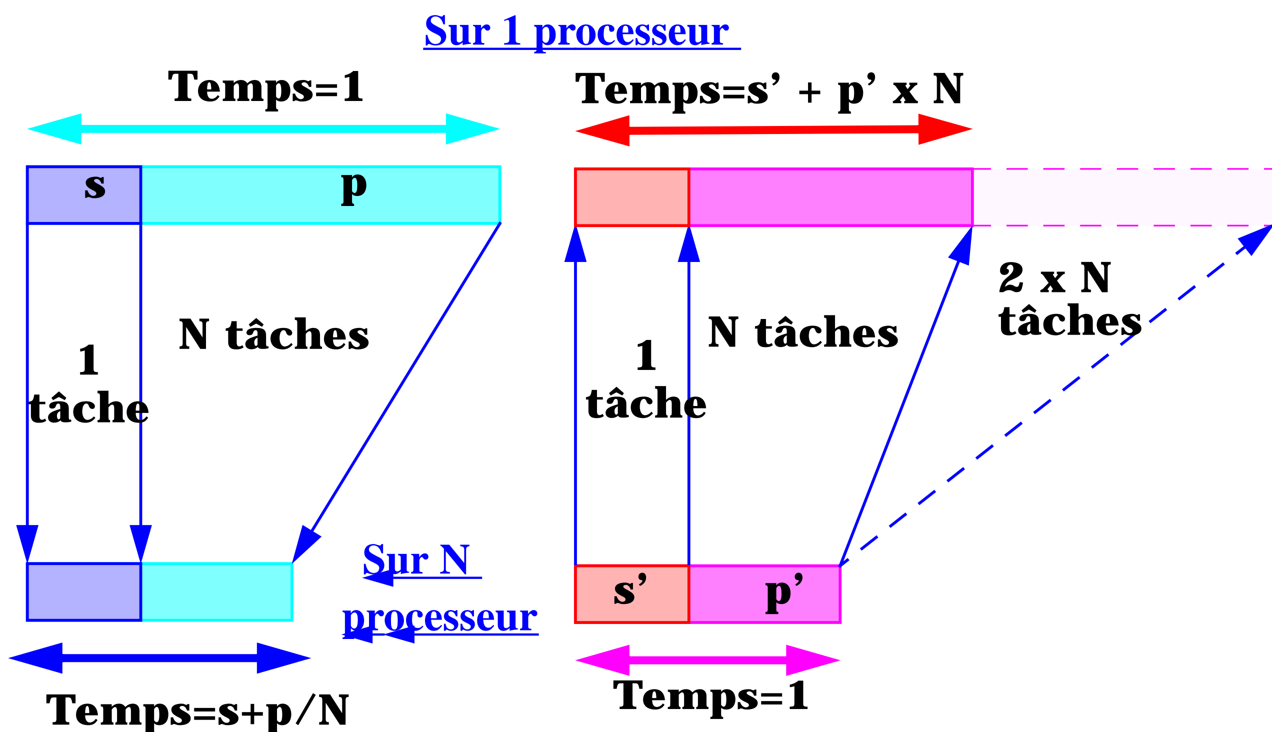


8.3 Limitations du parallélisme via OpenMP

8.3.1 Lois générales sur l'accélération d'un code parallèle

Loi d'Amdahl — **Modèle à taille fixe**

Loi de Gustafson — **Modèle à taille variable**



Accélération :

A = (s+p) / (s+p/N)

= 1 / (s+p/N)

-------> 1/s
N--> ∞

A = (s' + p' x N) / (s' + p')

= s' + p' x N

= N + (1-N) x s'

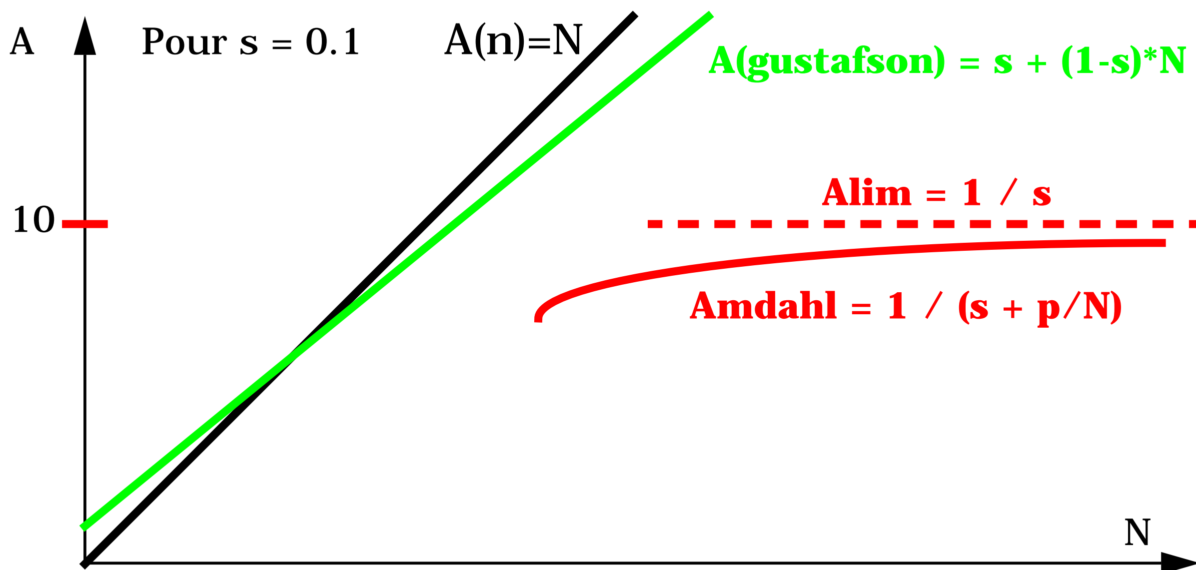
= N x (1-s') + s'

☞ Avec la loi d'**Amdahl,**

- on cherche, à partir d'une application séquentielle, à connaître l'accélération de sa version parallélisée,
- l'**accélération** est **bornée** par **la fraction séquentielle** (**1/s** asymptote horizontale).

☞ Avec la loi de **John L. Gustafson,**

- on calcule l'accélération d'une application déjà parallèle par rapport à une exécution sérialisée de celle-ci,
- l'**accélération** est **limitée** par l'asymptote **oblique** : seule la pente de celle-ci dépend de la **fraction séquentielle** .



☞ La loi de Gustafson (aussi appelée modèle à taille variable) sous-entend

- qu'il faut évaluer la fraction séquentielle sur une application déjà parallélisée,
- que l'on a un problème suffisamment gros pour avoir une accélération parfaite sur la partie parallèle.

8.3.2 Limitations propres à OpenMP

Ce qui n'est pas prévu dans le brouillon (*draft*) OpenMP et qui est laissé à l'appréciation des fournisseurs d'implémentations (constructeurs informatiques ou ISV) :

☞ La parallélisation sur architecture distribuée.

☞ La parallélisation **automatique**.

☞ Taux d'accélération (temps de restitution / temps CPU) non **garanti** ni **reproductible**; tout dépend du type d'exploitation

- dédié ou non,
- "*family*" ou "*gang scheduling*",
- i.e. le nombre de processeurs n'est pas forcément garanti.

☞ Outils et bibliothèques

- de débogage,
- de détermination (semi)automatique des attributs des variables (*autoscoper*),
- d'analyse des performances et surcoûts.
- Bibliothèques ***thread-safe*** et ***multithreadées***.

Domaines ardus ou intraitables avec OpenMP :

☞ Problèmes creux, maître-esclave.

☞ Programmation MPMD, client-serveur, calcul réparti ou collaboratif.

OpenMP

8.4 Outils et bibliothèques tierces

8.4.1 Les implémentations libres d'OpenMP.

Il en existe 2 :

☞ odinMP pour C/C++ uniquement,

☛ http://www.it.lth.se/odinmp/

☞ Omni 1.2 pour Fortran 77 ou C (grain fin seulement),

☛ http://pdplab.trc.rwcp.or.jp/Omni/

8.4.2 Compilateurs et outils correspondants

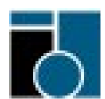
OpenMP : Outils d'aide au développement

	Fournissseur	Compilateur	paralléliseur automatique	autoscoper	débogueur	Moniteur de performances
Constructeur	Cray	f90		atscope dans xbrowse	Totalview	atexpert
	IBM	xlf90_r		Outils KAI recommandés		
	SGI	f90		?	Ladebug	speedshop
	NEC	f90	néant	néant	Totalview 4.0	Psuite
ISV	KAI	Guide	Visual pro	Guideview	Assure	Guideview
	PGI	pgf90		?	pgdbg	pgprof
	Pallas	néant				Vampir ?
	Etnus	néant			totalview 4.0	néant

☞ Suite KAPPRO de KAI[1]

☛ http://www.kai.com/parallel/kappro/platforms.html

1. Kuck & Associates Incorporated.



☞ Outils PGI de Portland Group :

- ☛ http://www.pgroup.com/ppro_docs

☞ Débogueur totalview d'ETNUS :

- ☛ http://www.etnus.com/products/totalview/index.html
- ☛ Bientôt disponible à l'IDRIS.

☞ Outil vampir de Pallas en cours de développement?

☞ Outils d'analyse des performances et surcoûts sur NEC.

- ☛ `Psuite`.
- ☛ option de *profiling* `-p`
- ☛ `ftrace`, bientôt.

8.4.3 Librairies parallèles compatibles avec OpenMP.

Attention aux librairies scientifiques qui ne seraient pas thread-safe (i.e qui auraient des effets de bord entre tâches).

☞ NAG[1]-Fortran version SMP 2.0 basée sur la version 19 de NAG-Fortran

- ☛ http:://www.c-s.fr

☞ NEC : bibliothèque ASL, sous-programmes *multithreadés* dont des FFT.

☞ Domaine public : bibliothèque PARBLAS basée sur les BLAS, elle est *multithreadée*.

1. Numerical Algorithm Group

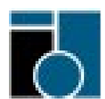
8.5 Evolutions d'OpenMP

La version 2 d'OpenMP qui est en cours de validation devraient être entérinée vers Septembre 2000.

8.5.1 OpenMP 2 (**)

Les évolutions les plus notables d'OpenMP 2 sont centrées sur l'interfaçage avec Fortran 95 :

- ☞ Directive **WORKSHARE** : parallélisation de notations F90 intrinsèquement parallèles :
  - ✒ **WHERE.**
  - ✒ **FORALL.**
  - ✒ **Les notations tableaux.**
- ☞ Directive **THREADPRIVATE :** privatisation de données modulaires ou rémanentes (**SAVE**).
- ☞ Clause **NUM_THREADS** : pour la directive **PARALLEL.**
- ☞ Clause **REDUCTION** : réductions sur des tableaux.
- ☞ Clause **COPYPRIVATE** de la directive **SINGLE** : *broadcast* de données vers les autres tâches.
- ☞ Sous-programmes OpenMP de *timing*.
- ☞ Verrous imbriqués.
- ☞ Contrôle du nombre de tâches pour le parallélisme multi-niveaux.



8.5.2 Evolutions à plus long terme,

elles pourraient être liées à des :

☞ entrées/sorties "globales mais partagées",

☞ directives de **distribution** des données (unification de HPF et OpenMP ?!).

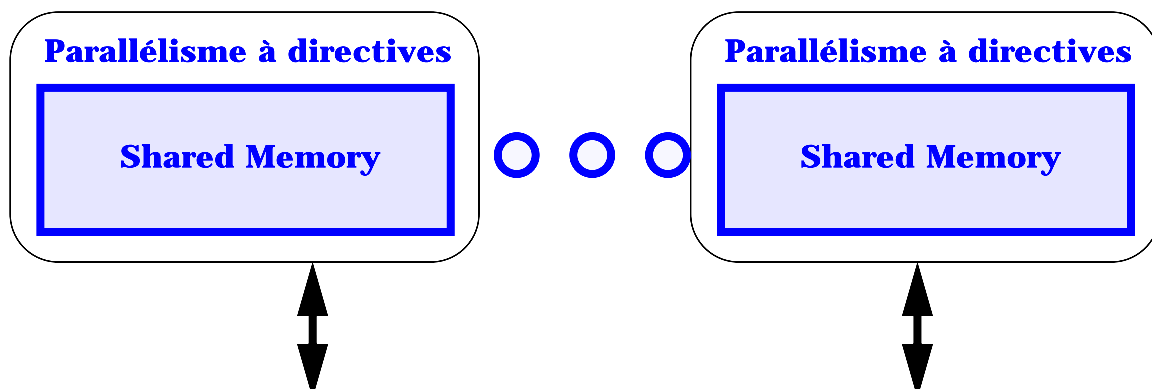
☞ directives de placement des tâches (Architecture NUMA) sur les processeurs ou groupe de processeurs quant à eux réellement UMA.

En effet, une tendance très nette se dégage pour une architecture à 2 niveaux des machines parallèles.

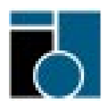
☞ *Clusters* de noeuds à mémoire partagée.

☞ Ce concept induit 2 niveaux de parallélisme :

- bas niveau à grain fin sur un noeud à mémoire partagée,
- niveau supérieur à gros grain sur une mémoire distribuée.



Passage de messages (MPI) ou HPF
ou directives propriétaires de placement?



ANNEXES

ANNEXE - A : synoptique OpenMP 2.0

Région parallèle

!$OMP [END] PARALLEL [SHARED(list)] [FIRSTPRIVATE(list)] [IF (logical)] [DEFAULT(NONE|PRIVATE|SHARED)] [COPYIN(list)] [NUM_THREADS]

Structure des données et partage du travail:

!$OMP [END] DO [SHARED(list)] [FIRST|LAST]PRIVATE(list)] [REDUCTION(operator|intrinsic, list)] [SCHEDULE(MODE[,chunk])] [ORDERED] [NOWAIT]

!$OMP [END] SECTIONS [[FIRST|LAST]PRIVATE(list)] [REDUCTION(operator|intrinsic, list)] [NOWAIT]
!$OMP SECTION
...
!$OMP [END] WORKSHARE

!$OMP THREADPRIVATE(/commun1/)

Les synchronisations

!$OMP BARRIER
!$OMP [END] SINGLE [NOWAIT] [COPYPRIVATE]
!$OMP [END] CRITICAL

!$OMP [END] MASTER
!$OMP [END] ORDERED

La *Run-time Library*

call OMP_SET_NUM_THREADS(N)
integer :: OMP_GET_THREAD_NUM
integer :: OMP_GET_NUM_PROCS
call OMP_SET_DYNAMIC()

integer::OMP_GET_NUM_THREADS
logical:: OMP_IN_PARALLEL
OMP_GET_MAX_THREADS
OMP_SET_NESTED()

Les variables d'environnement (en *Korn-Shell* sans export)

OMP_NUM_THREADS=4
OMP_DYNAMIC=TRUE

OMP_SCHEDULE="mode[,chunk]"
OMP_DYNAMIC=TRUE

OpenMP

ANNEXE - B : Directives C/C++

☞ La sentinelle est de la forme ***#pragma omp***

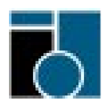
☞ Il n'y a pas de directive de fin de construction OpenMP; la portée lexicale d'une zone OpenMP est définie par **{ ... }**

☞ Les directives et clauses ont presque toutes un nom identique par rapport à Fortran exceptée la construction ***DO*** appelée ***for*** en C/C++.

```
#pragma omp for
{
  for (i=0; i<N; i++)
    {
     ...
    }
}
```

☞ En C/C++, il n'y a qu'une façon de faire de la compilation conditionnelle :

- ☛Par la pseudo-variable OpenMP prédéfinie.
- ☛Il n'y a pas d'équivalent C de la sentinelle **!$** de compilation conditionnelle.



*ANNEXE - C : Verrous **

- Pour une gestion plus fine des synchronisations
  - bloquer ou relâcher un PE,
  - définir des zones séquentielles,
  - protéger des variables partagées.

- **Le verrou doit être une variable partagée**
  - verrou = 1 <==> blocage
  - verrou=-1 <==> relâchement
  - verrou = 0 <==> verrou non initialisé

Verrou avant	Commandes	Verrou après	Action
0	call **omp_init_lock**	-1	On passe
1	call **omp_set_lock**(verrou)	1	Attente
-1		1	On passe
1	call **omp_unset_lock**(verrou)	-1	On passe
-1	booleen=**omp_test_lock**(verrou)	1	*TRUE*
1		1	*FALSE*
1 ou -1	call omp_destroy_lock(verrou)	0	On passe

Attention

- Ne pas se baser sur la valeur directe du verrou mais toujours utiliser **OMP_TEST_LOCK** pour connaître sa valeur !
- De plus, il est indispensable d'initialiser le verrou par un **OMP_INIT_LOCK** avant de l'utiliser. Un verrou étant partagé, une seule et unique tâche peut le faire.

Exemple AQ : section critique

```
!$OMP PARALLEL SHARED(iverrou)
!$OMP MASTER
  call omp_init_lock(iverrou)
!$OMP END MASTER
  call omp_set_lock(iverrou)
    Zone critique ...
  call omp_unset_lock(iverrou)
!$OMP END PARALLEL SHARED(iverrou)
call omp_destroy_lock(iverrou)
```

Exemple AR : section critique avec interruption

```
call omp_init_lock(iverrou)
!$OMP PARALLEL SHARED(iverrou)
  moi = omp_get_thread_num()
  call omp_set_lock(iverrou)
  print *,' Je suis le thread ',moi
  call omp_unset_lock(iverrou)

  DO WHILE  (.not. omp_test_lock(iverrou)) then
    !--- travail de recouvrement  avant de pénétrer en zone critique.
  END DO

  Zone critique ...
  print *,'tache num : ',moi,'travaille.'

  call omp_unset_lock(iverrou)
!$OMP END PARALLEL
call omp_destroy_lock(iverrou)
```

Autres intérêts :

☞ Eurékas : dès qu'une tâche a trouvé le résultat, elle dit aux autres d'arrêter (décryptage).

*ANNEXE - D : Comportement dépendant des implémentations **

Précisons que depuis OpenMP 2.0, chaque implémentation doit fournir une documentation précisant notamment ces comportements.

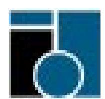
☞ Si les **ressources sont insuffisantes** pour créer les N tâches demandées, une implémentation peut au choix :

- ☛opter pour un arrêt brutal du programme (IBM NH1),
- ☛opter pour la création d'un plus petit nombre de tâches que précisé et ceci **même** dans le **mode statique** de création des régions parallèles (NEC SX5),
- ☛d'où l'intérêt de vérifier le nombre de tâches réellement obtenues par le sous-programme `omp_get_num_threads` ().

☞ Une mise à jour atomique peut être remplacée par une zone critique (**CRITICAL**).

☞ **L'ordonnancement** par défaut des **boucles partagées** (directive **DO**) n'est pas imposé par le standard.

- ☛Contrairement à une idée reçue, c'est souvent le mode **dynamique** et non le mode **statique** qui est choisi (ex : IBM).
- ☛i.e s'il n'est précisé ni par la variable d'environnement **OMP_SCHEDULE** ni par la clause **SCHEDULE(*MODE*)** de la directive **DO.**



OpenMP

☞ La **taille initiale des paquets** en ordonnancement **GUIDED**

```
export OMP_SCHEDULE="GUIDED,chunk)
```

- ***chunk*** ne désigne que la plus petite taille de paquet à l'exception du dernier).

☞ Le **Nombre de tâches par défaut** n'est pas normalisé.

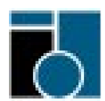
- i.e. s'il n'est précisé ni par la variable d'environnement **OMP_NUM_THREADS** ni par le sous-programme **omp_set_num_threads(N)** de la ***Run-time Library*** ni par la clause **NUM_THREADS(N)** de la directive **PARALLEL.**

☞ Le **mode par défaut** de **création des régions parallèles**

- s'il n'est précisé ni par la variable d'environnement **OMP_DYNAMIC** ni par le sous-programme **omp_set_dynamic(*logical*).**

☞ Le statut d'allocation des **tableaux dynamiques** s'ils n'ont pas été affectés par la clause **COPYIN** dans le **mode dynamique** de **création des régions parallèles,** est dépendant de l'implémentation.

☞ Le nombre de tâches pour chaque sous-région parallèle en cas de parallélisme imbriqué (**nesting**) n'est pas défini.



ANNEXE - E : Exemple de travail NQS sur NEC SX5

Exemple AS : Travail NQS type sur NEC SX5

```
#@$-q multi        # Option specifique IDRIS
#@$-lM 10Gb        # 10Go Mémoire
#@$-lT 36000       # 10h de temps CPU pour tout le job
#@$-lt 35900       # 10h-100 secondes CPU pour les process
#@$-eo -jo         #  eo: fusion des output standards et d'erreurs, jo: comptabilité
#@$ -c 8           #  8 processeurs maximum
#@$ -lu 9          #  9 tâches physiques maximum concurrement
                   #  En specifier une de plus que de tâches logiques.
set -x             #  Echo des commandes
cd $TMPDIR

cp $HOME/rep_openmp/mon_prog.f90 .
f90 -Popenmp -Wf"-ompctl condcomp" -R5 mon_prog.f90

export OMP_NUM_THREADS=8              # 8 tâches logiques
export OMP_DYNAMIC=FALSE              # Désactivation de mode dynamique des
                                      # régions parall'eles
export OMP_SCHEDULE="STATIC"          # Mode statique de répartition des itérations
                                      #  sur les tâches (défaut=DYNAMIC)

./a.out

cp mes_resultats.output $HOME/rep_openmp
ls -lrt
```

☞ La partie NQS est spécifique à la NEC SX5, les variables d'environnement par contre répondent au standard OpenMP. Signalons que le positionnement des 2 variables d'environnements **OMP_DYNAMIC** et **OMP_SCHEDULE** permet de se placer dans les modes d'ordonnancement les plus simples et moins coûteux.

*ANNEXE - F : Passer du macrotasking à OpenMP **

Exemple AT : code *macrotaske* (Cray ou NEC)

```
      do i=1,nproc-1
         itask(1,i)=3                     !---- Nombre d'esclaves
         itask(3,i)=i                     !---- Numero de la tache esclave
      enddo

C---  On lance 3 tâches esclaves  de numeros 1 a n-1.
     do iproc=1,nproc-1
        call tskstart(itask(1,iproc), sub_program_name,
     1  arg1, arg2, ..., domain(iproc), ... , argn)
     enddo

C---  Le maitre ( tâche numéro n) fait aussi sa part de travail.
      call  sub_program_name(arg1, arg2, ...,domain(nproc), ... , argn)

C---  Le maitre attend la fin des 3 taches esclaves.
      do   i=1,nproc-1
           call tskwait(itask(1,i))
     enddo
```

Exemple AU : code OpenMP (*domain decomposition*)

```
!$ integer :: omp_get_thread_num, omp_get_num_threads
...
!$OMP PARALLEL PRIVATE(iproc)
!$ iproc = omp_get_thread_num(); nproc = omp_get_num_threads()

!--- Attention au iproc +1 dans l'appel de cpyrot.
call sub_program_name(arg1, arg2, ...,domain(iproc+1), ... , argn)
!$OMP END PARALLEL       !--- Synchronisation implicite
```

☞ En OpenMP, la numérotation des tâches est de 0 à n-1.

OpenMP

ANNEXE - G : Lexique général

API : *Application Programmer Interface.*

ARB : *Administration Review Board*, organisme d'intérêt public propriétaire du "standard" OpenMP et qui veille à son évolution.

CPU : *Central Processor Unit*, on parle de temps CPU considéré comme le temps de travail effectivement consommées par les unités du (des) processeur(s) ou tourne l'application.

Data-parallelism : paradigme de programmation parallèle exécutant le même flot d'instructions sur des données différentes.

elapsed time : temps de restitution, aussi nommé *wall-clock time.*

HPC : *High Performance Computing.*

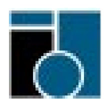
HPF : *High Performance Fortran*, autre "standard industriel" plus ancien qu'OpenMP datant de 1994, également basé sur des directives, il était par contre plutôt destiné à une programmation *data-parallel* sur architecture distribué.

ISV : Independant Software vendors. Fournisseurs de logiciels indépendants des constructeurs informatiques.

MPI - PVM : *Message Passing Interface - Parallel Virtual Machine*, bibliothèques d'échanges de messages pour la programmation d'applications parallèles.

overheads : surcoûts, dans ce livre coûts supplémentaires engendrés par l'utilisation de constructions OpenMP.

spin waiting : Attente active sur boucle. Lors d'un point de synchronisation les tâches peuvent se mettre à boucler sur celui-ci dans l'attente d'un événement



OpenMP

*ANNEXE - H : Lexique OpenMP **

Ce lexique est issu du *draft* OpenMP pour C et C++. Les termes officiels de la norme OpenMP restent en langue anglaise tandis que leurs définitions officielles approuvées par l'ARB sont librement traduites en langue française.

Barrier : Point de synchronisation devant être atteint par toutes les tâches de l'équipe, chacune attendant à ce point que toutes les autres soient également arrivées. Il y a les barrières explicites mais aussi implicites car intrinsèques à certaines constructions OpenMP.

Constructs : C'est une instruction qui consiste en au moins une directive OpenMP suivi d'un bloc structuré d'instructions sur lequel elle s'applique.

Dynamic extent : Toutes les instructions de la portée lexicale plus toutes les instructions exécutées dans des sous-programmes appelés au sein de la portée lexicale. On assimile la portée dynamique à la notion de région pour une construction (construct) OpenMP. En Français le terme de zone est moins ambigu que la notion de région parallèle qui est trop restrictive à la seule construction **PARALLEL/ENDPARALLEL**.

Lexical extent : Instructions lexicalement incluses dans un bloc structuré.

Parallel region : Instructions exécutées par les multiples tâches de l'équipe en concurrence.

Private : Accessibilité uniquement par une tâche de l'équipe dans une région parallèle. Il y a plusieurs façons de spécifier qu'une variable est privée selon son type.

Serial region : Instructions exécutées uniquement par la tâche maître hors de la région parallèle.

Serialize : Exécution d'une construction parallèle par une équipe (*team)* d'une seule tâche (en fait la tâche maître de cette construction parallèle) en respectant l'ordre d'exécution séquentiel (i.e comme s'il n'y avait pas parallélisation) des instructions à l'intérieur des blocs structurés. Dans ce cas **omp_in_parallel()** doit tout de même renvoyer la valeur qu'elle aurait renvoyé en cas de parallélisation effective.

shared : Accessibilité par toutes les tâches de l'équipe dans une région parallèle.

Structured block : Un bloc structuré est une instruction ayant une seule entrée et une seule issue. Une instruction ne constitue pas un bloc structuré s'il y a un branchement à l'intérieur ou vers l'extérieur de cette instruction; seul un appel a exit est *autorisé*. Une instruction composée est un bloc structuré si son exécution débute toujours à son début syntaxique et s'achève toujours à sa fin syntaxique. Une instruction: élémentaire, conditionnelle, itérative, ou d'essai (*bloc try* en C++) est un bloc structuré si les instructions la constituant (obtenu en les cernant par { et }) forment un bloc structuré. Les instructions de branchement, de labellisation ou de déclaration ne sont pas des blocs structurés.

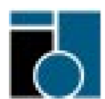
Variable : Un identificateur qui nomme un objet, optionnellement précédé d'un "espace de nom" (*namespaces*).

ANNEXE - I : Exercices récapitulatifs, corrections

- 1er exercice,
  - **SHARED** explicitement : aucune
  - **SHARED** implicitement :A,D,p,dyn1,work.tab, work.m, work.t
  - **PRIVATE** implicitement : work.local, work.autom, work.dyn, work.l, work.tmp
  - **PRIVATE** explicitement : B,C par THREADPRIVATE dans le main et le sous programme work.

- Explications pour quelques variables du 1er exercice.
  - `t` ayant été déclaré dans le sous-programme work avec l'attribut **SAVE**, elle est rémanente. Bien que déclarée localement dans un sous programme, une variables locale rémanente n'est pas empilée mais allouée dans la zone DATA qui est une zone globale pour toutes les tâches. De plus, elle n'est pas privatisable explicitement en OpenMP 1.X . Elle est donc implicitement **SHARED** à la différence de `local` qui est une variable locale non rémanente et donc implicitement privé.
  - `tab` étant un argument du sous-programme work, il hérite de l'attribut OpenMP passée en argument à l'appelant qui en l'occurrence est `A` qui est implicitement **SHARED**.
  - Le tableau dynamique `dyn2` doit être **privatisé** explicitement par la clause **PRIVATE** de la directive **PARALLEL**

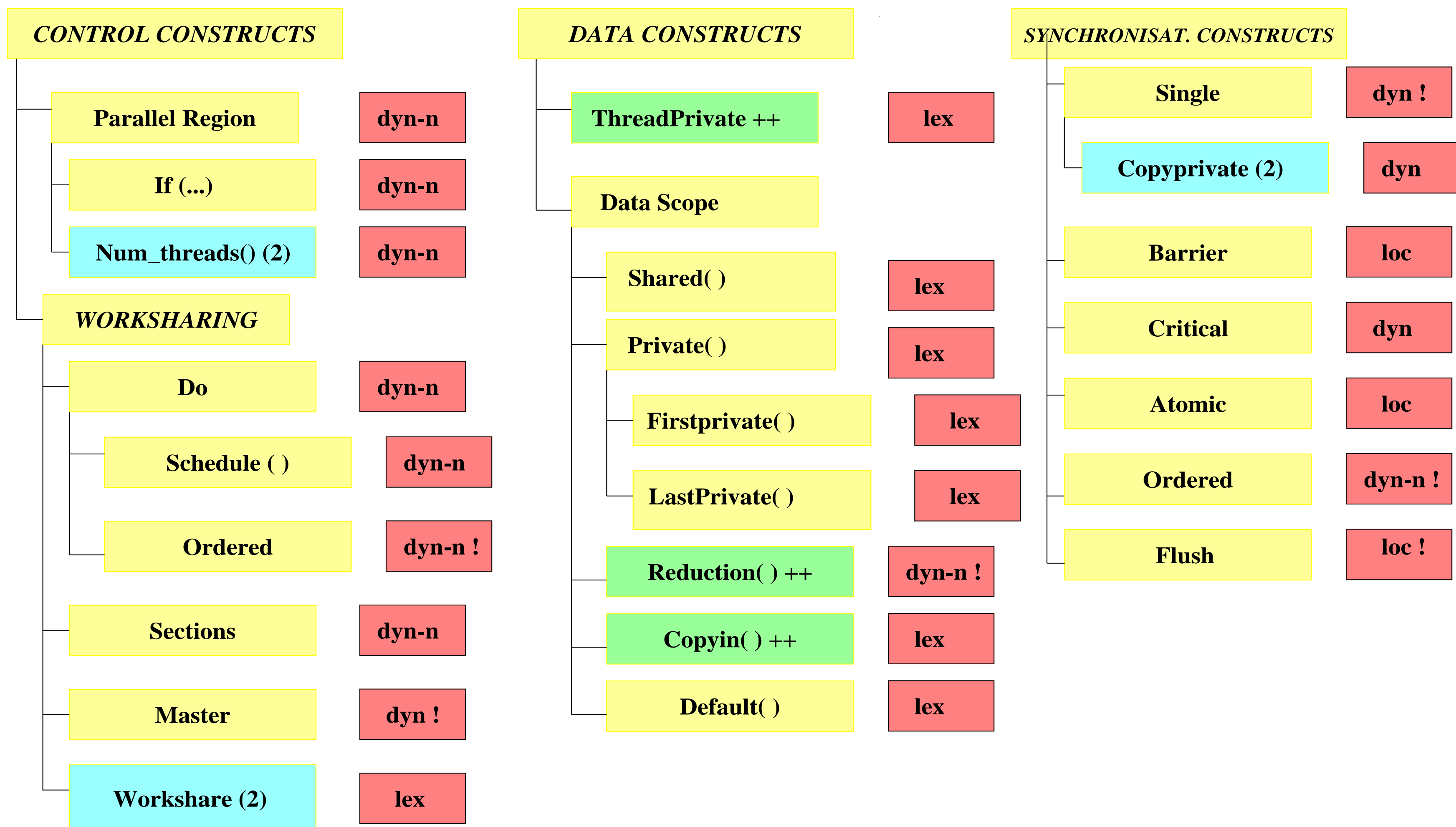
```
!$OMP PARALLEL PRIVATE(dyn2)
allocate(dyn2)
...
!$OMP END PARALLEL
```

Exercice récapitulatif. Correction

Determiner l'extention de chaque construction : locales (loc), lexicales (lex) ou dynamiques (dyn)

Questions subsidiaires : 1 - Indiquer (par dyn-n) les constructions dont l'extension dynamique est limitée par du NESTING
2- Quelles constructions ont ici une classification discutable (mettre ! dans les carrés rouges)



A

accélération 127
adressage indirect 81
Amdahl 127
ANSI 17
API 16, 102, 112, 143
ARB 15, 16, 143, 144
architecture 22, 110
architectures 17, 21
ASL 130
association 57
asymptote 127
ATOMIC 33, 67, 75, 80, 81, 105, 110
Attribut 48
attribution privée explicite 56
attribution privée implicite 56
autotasking 123

B

BARRIER 33, 58, 75, 86, 135
Barrier 144
barrières explicites 78
barrières implicites 78
Basic Storage Space 43
bench 15
beowulf 16
bibliothèque 25
bibliothèques 17, 23, 128, 129
binding 25
BLAS 130
bloc de contrôle 68
Block Started by Symbol 43
boucles éternelles 125
broadcast 77, 131
brouillon 15, 128
BSS 43, 44, 45

C

C$OMP 35
C/C++ 136
cache 111
calcul réparti 128
chunk 25, 94, 95, 96, 97, 98, 99, 115, 135, 140
Clusters 132

coarse-grain 17
cohérence 87
common 39, 43, 50, 52, 53, 81, 112
communication implicite 125
compilateur 22, 25, 37, 45, 46, 51, 111
Compilateurs 129
Compilation 27
compilation 37, 43, 125
compilation conditionnelle 36, 136
compliant 25
Comportement 139
concept 33, 125, 132
concurrent 63
Conflits 112, 125
consommateur 87
construct 25, 144
construction 25, 29, 33, 34, 48, 68, 77, 79, 81, 82, 83, 87, 91, 99, 110, 136, 144, 145
contentions 112
contexte 118
contexte d'appel 103
Contrôle 131
COPYIN 32, 53, 54, 59, 135, 140
COPYPRIVATE 76, 77, 131, 135
CPU 30, 75, 117, 143
Cray 16, 18, 27, 142
création 32, 139, 140
creux 128
CRITICAL 58, 75, 79, 86, 103, 105, 110, 135, 139

D

DATA 59, 60
Data-parallelism 143
deadlock 125
débit 112
Débogage 75
décomposition de domaines 16, 17, 51, 60, 108, 118, 119
décryptage 138
DEFAULT 32, 51, 53, 135
DEFAULT(NONE) 51
dégradation 111
descripteurs 28
déséquilibre 96
Diffusion 77
directives 22, 35
distribuée 128, 132
distribuées 16
distribution 132

DO 17, 50, 64, 65, 66, 70, 75, 78, 82, 83, 84, 86, 93, 94, 95, 96, 97, 99, 105, 106, 107, 111, 135, 136, 139
do while 66, 107
documentation 116, 139
domain decomposition 142
draft 15, 17, 59, 83, 128, 144
durée de vie 39, 40, 48
DYNAMIC 94, 95, 96, 99, 106
Dynamic 144

E

Effets de courses 125
elapsed time 117, 143
élémentaire 68
elemental 68
Entrées/Sorties 43
entrées/sorties 132
équilibrage 97
équivalences 81, 83
Eurékas 138
Evolutions 131, 132
exclusion mutuelle 79
export 27, 100, 135
extensible 63
extension 25
extent 25, 144

F

FFT 106, 130
FIRSTPRIVATE 32, 50, 53, 63, 65, 76, 92, 135
FLUSH 33, 75, 86, 87, 88
fonctions intrinsèques 67, 91
FORALL 67, 68, 91, 131
fork and join 29
Format fixe 35, 36, 37
Format libre 35, 36, 37
Fortran 15, 36, 60, 91, 111, 136
Fortran 95 15, 50, 57, 67, 91, 116, 131
FORUM 17
fournisseurs 102

G

Gestion mémoire 39
ghost-cells 109

grain fin 16, 123, 132
granularity 25
gros grain 16, 17, 123, 132
GUIDED 94, 115, 140
Gustafson 127

H

heap 44
hiérarchie mémoire 87
HPC 18, 143
HPF 20, 132, 143

I

IBM 16, 27, 28, 30, 101, 104, 139
identificateur 145
IF 29, 32, 71, 111, 135
imbrication 104, 120
implémentation 15, 25, 102, 120, 139, 140
implémentations 30
implicit none 51
indices de boucle 49
Input/Output 43
interfaçage 91, 131
interface explicite 92
interprétations 16
intrinsic 80, 82
ISV 143

J

Jacobi 107

L

LASTPRIVATE 50, 53, 63, 65, 66, 92
Lexical extent 144
librairies scientifiques 130
Limitations 91, 126, 128
livelocks 125
Loop-level parallelism 106, 110, 112, 123

M

macro 36, 37
macrotasking 142
maître-esclave 128
majuscules 36
MASTER 72, 105, 135, 138
MATMUL 67, 71, 91
mémoire virtuellement partagée 16, 112
mesure du temps 117
Méthodologie 109, 123
minuscules 36
mise à jour atomique 80, 93, 139
mode dynamique 52, 95, 100, 101, 115, 118, 120, 139, 140, 141
mode guided 97, 99
mode stack 45
mode statique 45, 94, 139
modèle 17, 20, 28, 29, 123, 125, 127
modules 39, 57, 59, 91
MPI 17, 20, 143
MPMD 128
multiprocesseurs 16
multithreadés 16

N

namespaces 145
NEC 16, 27, 28, 30, 103, 111, 130, 139, 141, 142
nesting 25, 104, 105, 120, 140
Niveaux de répartition 102
notations tableaux 67, 91, 131
NOWAIT 63, 64, 65, 67, 69, 70, 71, 76, 78, 86, 107, 110, 135
NQS 141
NUM_THREADS 32, 71, 131, 135, 140
NUMA 112, 132
numérotation 31, 142

O

omp_destroy_lock 137, 138
OMP_DYNAMIC 100, 115, 135, 140, 141
OMP_GET_DYNAMIC 120
OMP_GET_MAX_THREADS 119, 135
OMP_GET_NESTED 120
OMP_GET_NUM_PROCS 119
OMP_GET_NUM_THREADS 118, 135
omp_get_num_threads 95, 99, 108, 139, 142

OMP_GET_THREAD_NUM 31, 118, 135
omp_get_thread_num 54, 108, 142
OMP_GET_WTICK 117
OMP_GET_WTIME 117
OMP_IN_PARALLEL 119, 135
omp_in_parallel 103, 145
omp_init_lock 137, 138
omp_lib 91, 116
OMP_NESTED 115
OMP_NUM_THREADS 23, 27, 32, 54, 100, 115, 135, 140, 141
OMP_SCHEDULE 94, 115, 135, 139, 140, 141
OMP_SET_DYNAMIC 120, 135
omp_set_dynamic 100, 140
omp_set_lock 137, 138
OMP_SET_NESTED 120, 135
OMP_SET_NUM_THREADS 32, 118, 135
omp_set_num_threads 23
omp_set_num_threads(140
omp_test_lock 137, 138
omp_unset_lock 137, 138
OpenMP 2.0 59, 83, 135, 139
opérateur 80
operator 82
optimisation 110, 125
ORDERED 65, 66, 75, 84, 85, 86, 105, 135
ordonnancement 94, 96, 98, 99, 102, 139, 140
orphaned 103
orphaning 25, 33, 103
outils 112, 124
overheads 75, 143

P

paquets 95, 110
PARALLEL 29, 32, 33, 34, 50, 59, 64, 70, 82, 86, 105, 107, 108, 131, 135, 140, 142, 144
PARALLEL DO 83, 94, 95, 96, 97, 106
PARALLEL SECTIONS 69
parallélisation automatique 123, 128
parallélisme de contrôle 17, 28
parallélisme de données 28
parallélisme imbriqué 140
PARBLAS 130
parentés 105
partage du travail 22, 50, 71, 135
pavage 98, 112
performances 75, 109, 110, 125, 128, 130
persistantes 55
pièges 125

pile 28, 44, 46
placement 102, 112, 132
pointeur d'instructions propres 28
pointeur de pile 28
pointeurs 50, 92
Poisson 107
portabilité 16, 22
portée 31
portée dynamique 31, 34, 48, 82, 84, 144
portée lexicale 31, 33, 34, 48, 49, 51, 56, 60, 63, 68, 136, 144
Portée locale 33
pragma omp 136
Précision 117
précompilateurs 37
préprocessing 36
PRIVATE 32, 36, 48, 51, 53, 61, 63, 65, 77, 92, 107
Private 145
privatisation 57
privatisés 49
processeurs 119
processus légers 25, 28, 46
producteur 87
programmation modulaire 103
ps 28
PSE 22
pseudo-variable 136
Psuite 130
public 130
PVM 17, 20, 143

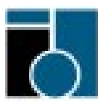
R

race conditions 125
REDUCTION 32, 33, 53, 63, 65, 75, 82, 83, 93, 107, 110, 131, 135
réduction 81
region 25, 144
Région parallèle 70, 71, 135
région parallèle 29, 31, 32, 33, 34, 48, 49, 51, 53, 55, 56, 60, 64, 69, 91, 92, 93, 100, 101, 103, 105, 115, 118, 140, 144, 145
région parallèle imbriquée 118, 120
régions parallèles 116
Régions parallèles dynamiques 100
registres 111
Règles 35
relâchement 137
relations 105
rémanentes 41, 59, 60
Restrictions 92

round-robin 95
RUNTIME 94, 110
Run-time Library 23, 31, 91, 100, 116, 135, 140

S

SAVE 41, 59, 60
scalable 63
SCHEDULE 65, 66, 94, 95, 96, 97, 99, 106, 111, 135, 139
SCHEDULE(RUNTIME) 110
scheduling 25
SECTION 63, 69, 135
section critique 80, 83, 138
SECTIONS 63, 66, 69, 75, 78, 82, 86, 105, 135
sentinelle 35, 36, 37, 136
Serial 145
sérialisation 104
sérialisée 118, 120
Serialize 145
SGI 16, 28, 112
shadow-buffer 109
SHARED 32, 33, 36, 48, 51, 53, 57, 58, 59, 61, 82, 108, 135, 138
shared 145
SINGLE 75, 76, 77, 78, 86, 105, 110, 131, 135
sleep 30
SMP 130
sous-processus 28
spin waiting 30, 75, 143
SPMD 123
stack 28, 44, 45, 46, 57
Stack Model 46
standard industriel 143
statements 25
STATIC 94, 111
statut d'allocation 140
statut des variables 93
Statut implicite 60
structure 123
Structured 145
SUM 67
SUN 16
surcoûts 75, 97, 110, 112, 128, 130, 143
SX5 16, 27, 28, 30, 103, 111, 139, 141
synchronisation 22, 88, 144
synchronisations 72, 87, 110, 135, 137
synoptique 135
syntaxiques 35



T

tableaux à profil implicite 50, 92
tableaux à taille implicite 50, 92
tableaux automatiques 41, 42, 44, 47, 60
tableaux dynamiques 42, 44, 50, 56, 91, 140
tas 44
team 25, 145
temps de restitution 16, 110, 117, 125, 128, 143
Terminologie 25
thread 56
THREADPRIVATE 33, 50, 52, 53, 54, 55, 57, 59, 77, 101, 131, 135
threads 23, 25, 28, 46, 118, 119, 120
tiling 98
timing 117, 131
tutoriaux 15

U

UMA 132
USER 111

V

Variable 145
variable d'environnement 32, 94, 100, 139, 140
variable locale 40
variables 60
variables automatiques 41
variables d'environnement 23, 31, 115, 116, 135, 141
variables locales 45, 47, 60
variables locales automatiques 40, 45
variables statiques initialisées 44
variables statiques non initialisées 44
vectorisation 111
Verrous 75, 131, 137
verrous mortels 125
vidage 86

W

wall-clock 143
wall-clock time 117
WHERE 67, 68, 131
WORKSHARE 67, 68, 71, 91, 105, 131, 135

Z

zone 25, 33
zone BSS 43, 44
zone critique 80, 81, 138, 139
zone DATA 43, 44
zone maître 72
zone MASTER 72
zone mémoire 44, 57, 83
zone SECTIONS 78
zone SINGLE 76, 77, 78
zone TXT 43
zone U 43
zones critiques 79, 83
zones globales 46
zones mémoires 43
zones ordonnées 85