

Formation théorique : environnement de travail sur le cluster IBM

Guy Moebs - CRIHAN



Plan de la formation

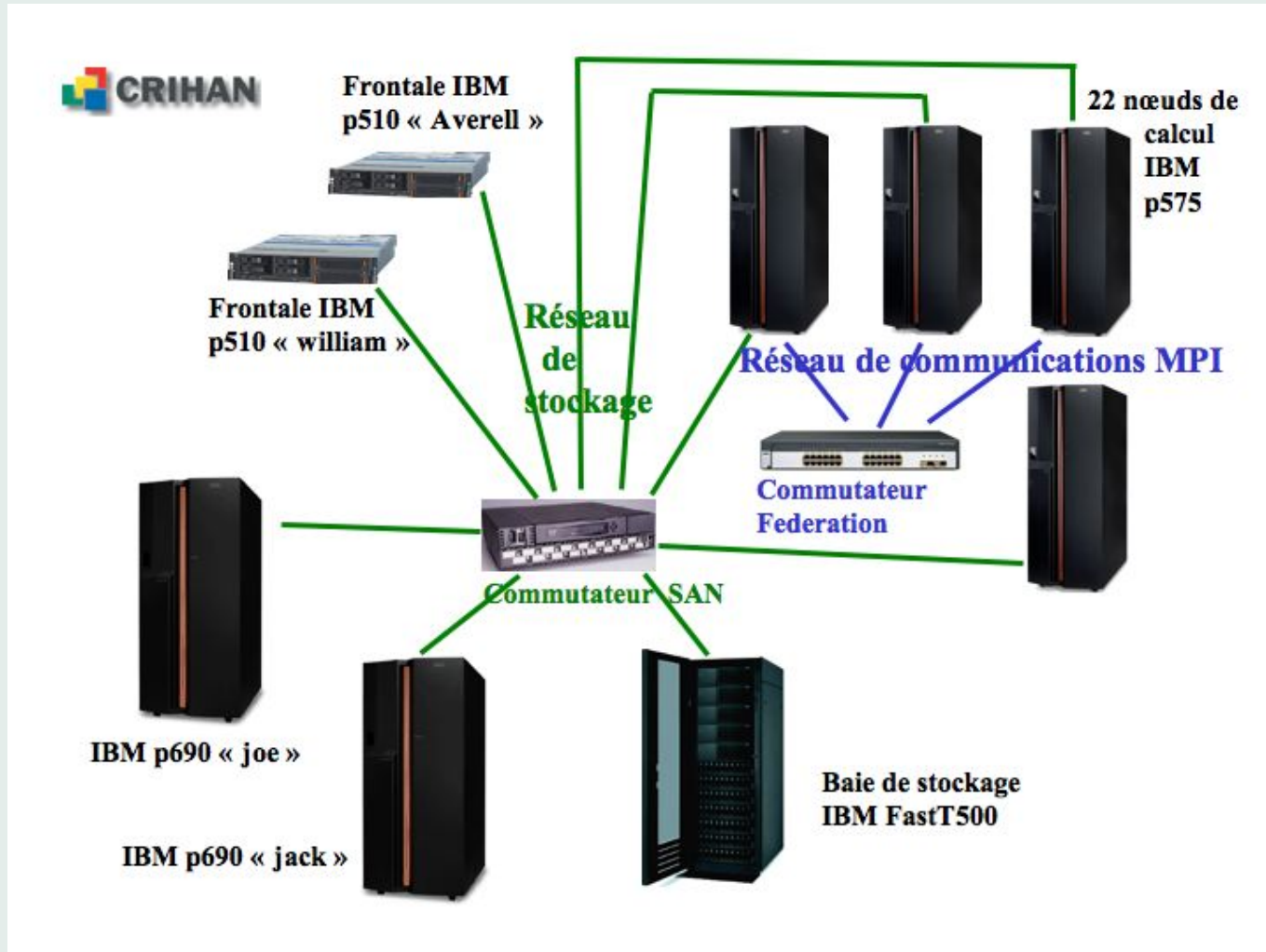
1. Descriptions matérielle et logicielle
2. Soumission des calculs au travers de LoadLeveler
3. Environnement de compilation XLF
4. Bibliothèques scientifiques
5. Outils (débugage & analyse)
6. Considérations matérielles
7. Optimisation scalaire
8. Calcul parallèle
9. Echange de messages avec MPI
10. Partage du travail avec OpenMP
11. Exemple récapitulatif

1. Descriptions matérielle et logicielle

1.1. Description matérielle

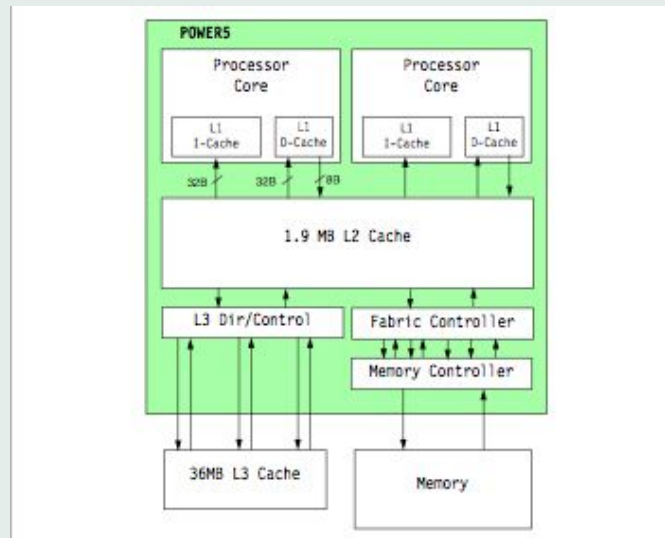
Le cluster IBM est composé de :

- 2 frontales de connexion p510, *william* et *averell*, dotées chacune de :
 - 2 processeurs Power 5 (1.5 GHz) ;
 - 4 Go de mémoire vive ;
- 22 nœuds de calcul SMP p575 dotés chacun de :
 - 8 processeurs Power 5 (1.9 GHz, 7.6 GFlops),
 - 16 Go de mémoire vive ;
- 1 commutateur à faible latence *Federation* qui interconnecte 16 des 22 nœuds de calcul ;
- 1 baie de disques FastT500 (20 To de disque utile), reliée à chacun des nœuds par des liens fibre optique (2Gb/s) ;
- les deux nœuds p690 seront intégrés à la solution dans les prochains jours.



1.2. La puce Power 5

- La puce Power5 est à double-cœur mais un seul est activé
- Il dispose d'une hiérarchie de caches L1, L2, L3



1.3. Le Multi Chip Module (MCM)

- Le bloc de base pour un nœud p575 est le MCM (Multi Chip Module)
- Il se compose d'une seule puce Power 5
- Chaque p575 contient donc 8 MCM

1.4. Latences (en cycles d'horloge) et associativité

	Power4	Power5
Registres	1	1
Cache L1	2	2
Cache L2	12	13
Cache L3	123	87
Mémoire	351	235

	Power4	Power5
Cache L1	2 voies associatives, FIFO 32 ko (D) + 64 ko (I) 128 octets	4 voies associatives LRU 32 ko (D) + 64 ko (I) 128 octets
Cache L2	4 voies associatives 1440 ko (par double cœur) 128 octets	10 voies associatives 1920 ko (un seul cœur) 128 octets
Cache L3	8 voies associatives 32 Mo 256 octets	12 voies associatives 36 Mo 256 octets

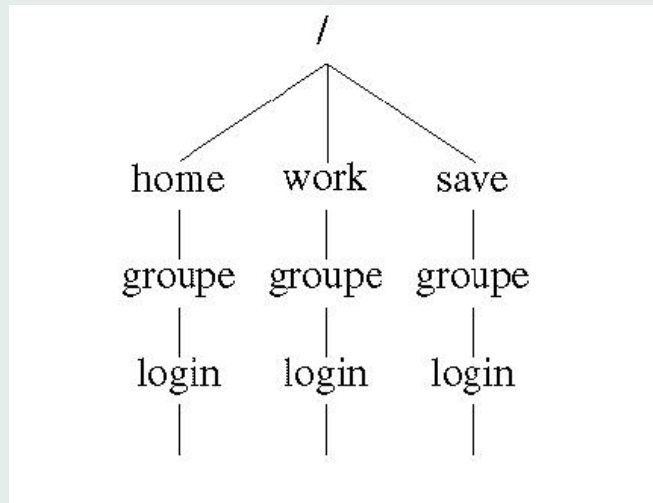
1.5. Configuration mémoire

- Chaque nœud de calcul p575 dispose de 16 Go de mémoire physique
⇒ soit environ 12 Go de mémoire disponible pour les applications
- Chaque nœud est configuré en 10 Go de *Large Pages* (LP) et 2 Go de *Small Pages* (SP)
- Une Large Page est une page mémoire de 16 Mo
- Une Small Page est une page mémoire de 4 ko
- Les performances sont nettement meilleures (jusqu'à 60% de gain) lorsque les *Large Pages* sont employées (Data Prefetch Streams, ..)
⇒ Elles sont mises par défaut sur tous les nœuds
- Qu'est-ce que cela change pour les utilisateurs ?
⇒ Rien ou presque sauf à l'édition des liens !
- Il faut aussi légèrement modifier les scripts de soumission
- Les *Large Pages* sont employées pour les données globales (heap)
- Les *Small Pages* sont employées pour les variables locales (stack)

1.6. Espaces utilisateurs

Chaque utilisateur dispose de 3 espaces permanents au sein de son projet qui sont physiquement localisés dans la baie FastT500.

- un espace utilisateur `/home/projet/login` (5 Go),
- un espace de travail `/work/projet/login` (25 Go),
- un espace de stockage `/save/projet/login` (300 Go).



⇒ Les exécutions des programmes en mode batch se font dans des espaces temporaires situés dans la baie de disques FastT500

1.7. Description logicielle

Système d'exploitation	AIX 5.3 , UNIX d'IBM
Compilateur Fortran	XLF, v. 9.1.0.4
Compilateur C	XLC, v. 7.0.0.3
Bibliothèques scientifiques	ESSL (inclus BLAS, FFT), v 4.2.0.0 ; LAPack, v 3.0 ; P-ESSL, v 3.2.0.0
Calcul parallèle	MPI, POE v 3.2.0.20 ; OpenMP
Gestionnaire de batch	LoadLeveler, v. 4.2.2.1
Outils d'analyse	prof, gprof, xprofler, PE Benchmark (pct, pvt, ute), jumpshot
Débogueurs symboliques	dbx, pdbx
Editeurs de texte	vi ; emacs, v 21.4.1
Graphisme	gnuplot, v 3.7.2 ; xmgrace, v 5.1.11 ; pgplot ; ncview
Bibliothèque	netCDF (32 et 64 bits), v 3.6.0-p1 ; NCO v 3.0.0

1.8. Logiciels scientifiques de chimie disponibles

Codes commerciaux accessibles à tous les utilisateurs

Gaussian G03, G98

Schroedinger Jaguar 6.0, 5.0, 4.1

Accelrys Catalyst 4.9, CNX 2002

Codes "libres" sous licence (usage non commercial)

IBM CPMD v 3.9.1

Yale Univ. CNS v 1.1 (version gratuite de CNX)

Iowa State Univ. Gamess 16/02/2002

- ⇒ On peut installer vos logiciels commerciaux sous licence :
- fournissez-nous vos licences ...
 - on restreint les droits d'accès à votre projet



1.9. Connexion et modes d'exécution

- Les ressources du cluster sont séparées en deux parties :
 - (i) **les frontales de connexion**, bi-processeurs, p510, pour la connexion et la petite mise au point ;
 - (ii) **les nœuds de calculs**, dédié au batch : 22 nœuds octo-processeurs p575 et 2 nœuds p690.
- Les connexions, les éditions, les compilations, les exécutions en interactif prennent leurs ressources dans la partie (i)
- L'utilisateur exprime ses besoins en temps, processeurs, mémoire (data + stack), dans un script
- Chaque nœud p575 dispose de 16 Go de mémoire physique, soit environ 12 Go de mémoire disponible pour les applications
 - ⇒ Sa mémoire est configurée en 10 Go de *Large Pages* (LP) et 2 Go de *Small Pages* (SP)
- Chaque nœud p690 dispose de 32 Go de mémoire physique, soit environ 30 Go de mémoire disponible pour les applications
 - ⇒ Sa mémoire est configurée en 20 Go de *Large Pages* (LP) et 10 Go de *Small Pages* (SP)

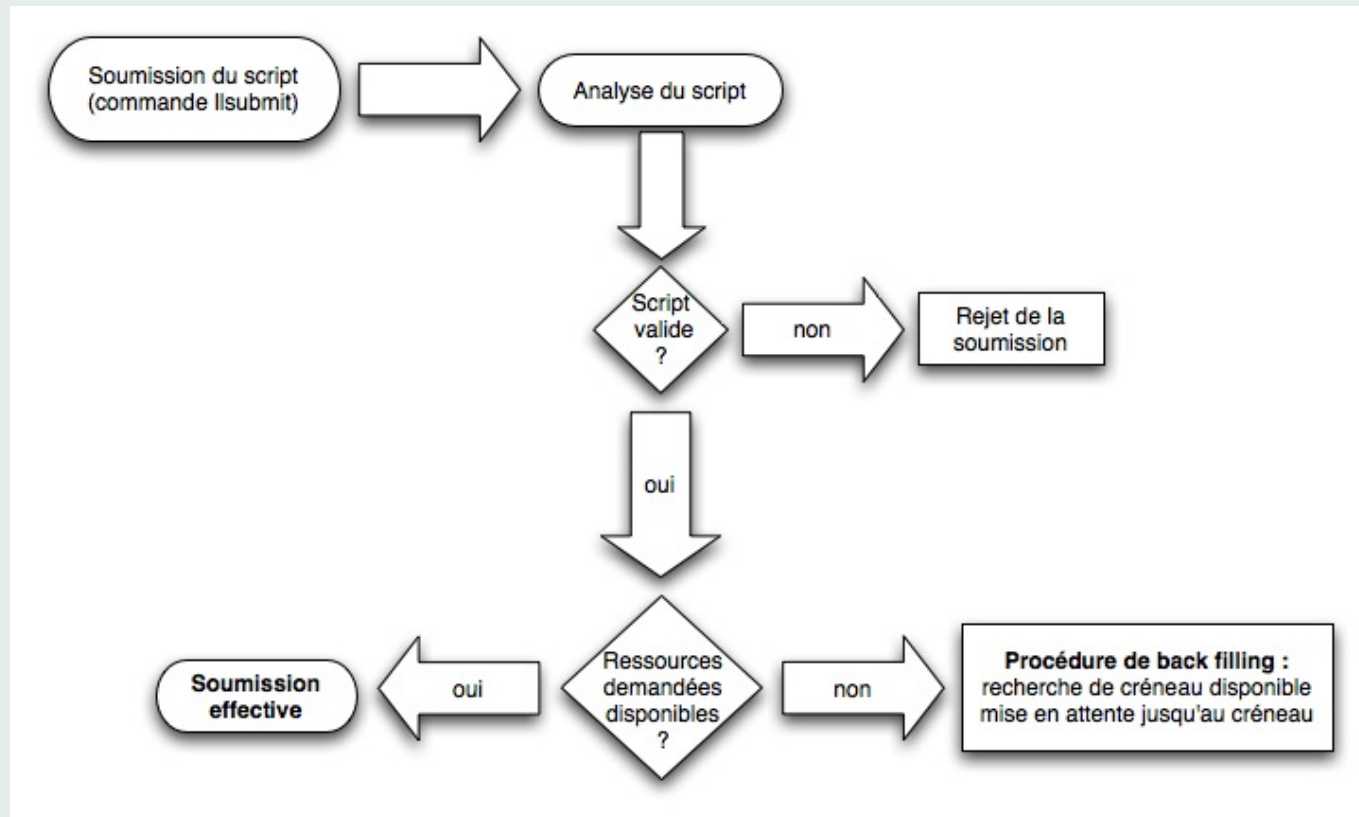
2. Soumission des calculs au travers de LoadLeveler

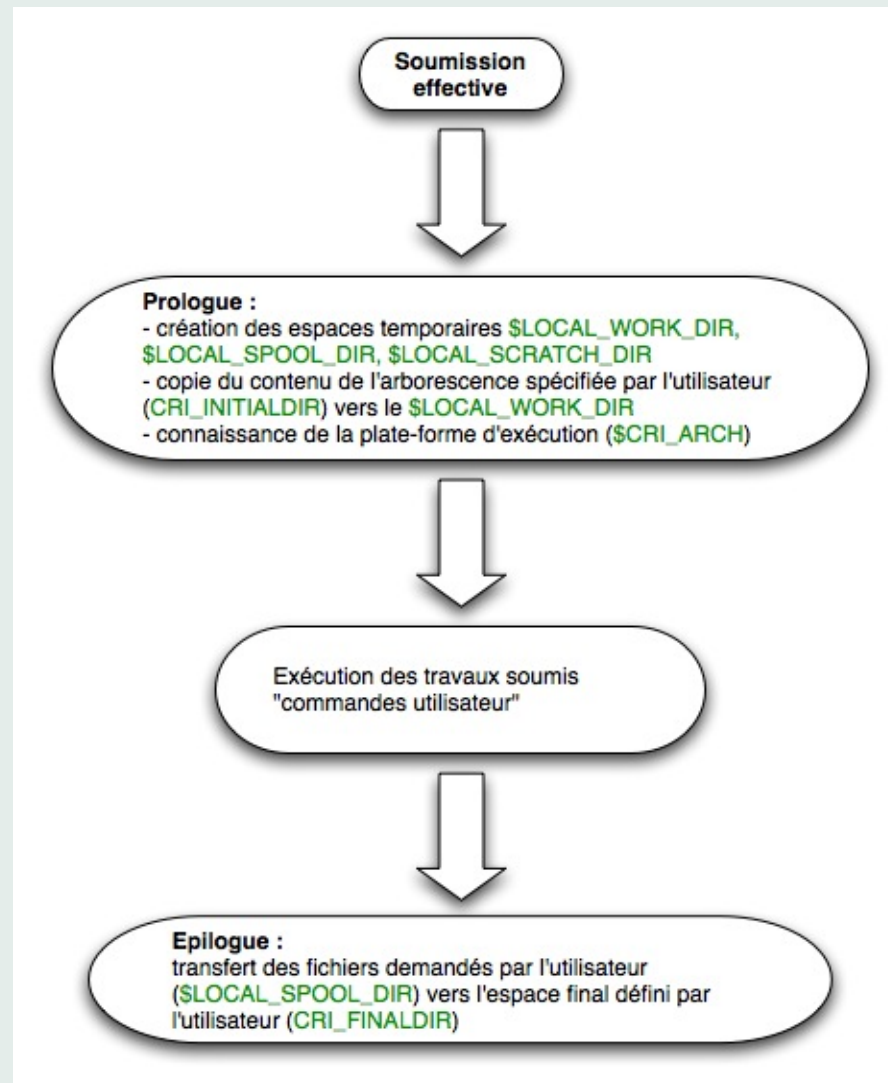
- Loadleveler : logiciel de soumission de travaux sur calculateur IBM
- La soumission fonctionne à l'aide de scripts
- Les paramètres sont transmis à l'aide de directives
- les directives ont la forme # @ **directive**
- Il n'y a pas de queue batch sur le cluster
- L'expression des besoins "pondère" les soumissions et décide des priorités

2.1. Spécificités architecturales

- Les comptes utilisateurs sont situés dans la baie externe FastT500
- Les exécutions ont lieu dans les disques de la baie externe (/dlocal)
 - ⇒ les jobs MPI sont à cheval sur les nœuds par défaut
 - ⇒ Le système gère les procédures de transfert de fichiers
- Il faut fixer les quantités de mémoire allouées pour les *Large Pages* et les *Small Pages*

2.2. Principe des soumissions





- Le **\$LOCAL_WORK_DIR** représente le répertoire temporaire créé par le système dans les disques de la baie FastT500, visible par tous les nœuds d'exécution ;
→ /dlocal/run/ll-william.xxxx.0 ou /dlocal/run/ll-averell.xxxx.0
- Le **CRI_INITIALDIR** représente le répertoire spécifié par l'utilisateur dans son espace permanent sur la baie FastT500 dont le contenu est dupliqué dans le **\$LOCAL_WORK_DIR** ;
→ /work/projet/login/REPertoire_ENTREE
- Le **CRI_FINALDIR** représente le répertoire spécifié par l'utilisateur dans son espace permanent sur la baie FastT500 dans lequel les fichiers spécifiés par l'utilisateur sont rapatriés par le système ;
→ /work/projet/login/REPertoire_SORTIE
- Le **\$LOCAL_SPOOL_DIR** représente le répertoire temporaire créé par le système dans les disques de la baie FastT500 qui sert de transit avant le rapatriement vers l'espace permanent de l'utilisateur situé dans la baie, visible par tous les nœuds d'exécution.
→ /dlocal/spool/ll-william.xxxx.0 ou /dlocal/spool/ll-averell.xxxx.0

- Le **\$LOCAL_SCRATCH_DIR** représente le répertoire temporaire créé dans les disques internes de chaque nœud et pouvant servir de mémoire tampon si nécessaire
 - ⇒ Très utile pour les codes utilisant des gros volumes disques comme zone de scratch
 - ⇒ Espace d'environ 80 Go par nœud, partagé entre les jobs du nœud, sans contrôle (quota de 70 Go par utilisateur)
- Le **\$LOCAL_SCRATCH_DIR** n'est visible qu'au sein de son nœud
 - ⇒ Son usage est donc restreint aux applications s'exécutant en intra-nœud et aux fichiers gérés par un ensemble de processus dans un même nœud
- Les **CRI_INITIALDIR** et **CRI_FINALDIR** doivent obligatoirement respecter les règles suivantes :
 - être des chemins absolu dans le \$HOME_DIR ou le \$WORK_DIR
 - ne contenir que des fichiers réguliers
 - avoir des droits (lecture/écriture) compatibles avec les actions
 - le **CRI_INITIALDIR** ne doit pas être vide
 - le **CRI_FINALDIR** est créé s'il n'existe pas

2.3. Sur quels nœud(s) tournent mes jobs ?

- Plate-forme hétérogène
- Les jobs non spécifiques ⁽¹⁾ sont susceptibles de s'exécuter sur tous les nœuds
- Loadleveler attribue les ressources parmi ses disponibilités
- L'optimisation d'un code à une plate-forme améliore ses performances sur cette plate-forme mais
qu'en est-il sur une autre ???

- La compatibilité est ascendante :

{ un code optimisé pour Power 4 s'exécute normalement sur Power 5
 un code optimisé pour Power 5 **peut ne pas s'exécuter** normalement sur Power 4 !!

- Un code optimisé pour Power 4 est moins efficace sur Power 5 qu'un code optimisé pour Power 5 (dégradation variable, dépend de l'application)

⇒ Alors que faire ???

(1) jobs spécifiques : une seule architecture peut satisfaire les besoins en ressources, par exemple demander plus de 10 Go par processus envoi sur p690, demander plus de 32 processeurs en MPI envoi sur p575, ...

Sur quels nœud(s) tournent mes jobs ?

- Avoir un binaire, si possible, pour chaque plate-forme
 - ⇒ Avoir deux fichiers de compilation Makefile
 - ⇒ Avoir deux répertoires pour les fichiers objet
- S'adapter à la plate-forme d'exécution :
 - qarch=pwr4 -qtune=pwr4 pour les code destinés aux p690
 - qarch=pwr5 -qtune=pwr5 pour les code destinés aux p575
- Ajouter le choix du binaire, selon la plate-forme d'exécution, dans le script de soumission :

Script de soumission en Korn Shell

```
if [$CRI_ARCH = "pwr4"]
then
  ./a.out_ pwr4
else
  ./a.out_ pwr5
fi
```

Script de soumission en C Shell

```
if ($CRI_ARCH == pwr4)
then
  ./a.out_ pwr4
else
  ./a.out_ pwr5
endif
```

- Cela ne concerne pas les applications précompilées installées par le CRIHAN

2.4. Mots-clés

Mot-clé	Séquentiel	MPI	OpenMP	Gaussian (03,98)
<code>cri_job_type</code>	<code>serial</code>	<code>mpi</code>	<code>openmp</code>	<code>gaussian</code>
<code>cri_total_tasks</code>	sans objet	nombre de processus MPI	nombre de threads OpenMP	nombre de threads Gaussian

- `wall_clock_limit` : temps de présence du job sur la machine
- `data_limit` : mémoire par processus
 - ⇒ Mémoire prise dans les *Large Pages*
 - ⇒ Attention à l'option `-bmaxdata`
- `stack_limit` : mémoire par processus
 - ⇒ Mémoire prise dans les *Small Pages*
 - ⇒ Mot-clef optionnel (256 Mo par défaut)
- `core_limit` : taille maximale par processus
 - ⇒ Mot clef optionnel (0 Mo par défaut)
- Liste non exhaustive ... (il existe aussi des scripts pour Gamess, Jaguar, ...)
- à compléter selon les besoins des utilisateurs

2.5. Commandes

- soumission : `llsubmit script_job → ref_job` (ll-william.xxxx.0 ou ll-averell.xxxx.0)
`llsubmit ll_mpi`
`llsubmit: Processed command file through Submit Filter: "/soft/load/LLFILTER/llfilter"`.
`llsubmit: The job "william-a1.crihan.fr.2045" has been submitted.`

- suivi : `llq`

Id	Owner	Submitted	ST	PRI	Class	Running On
william-a1.1973.0	login	2/1 13:10	R	50	hps_indus	p5-a1-n23
william-a1.1848.0	login	1/30 09:21	R	50	hps_large	p5-a1-n7
william-a1.1899.0	login	1/30 20:51	R	50	medium	p5-a1-n24
william-a1.1978.0	login	2/1 14:28	R	50	xsmall	p5-a1-n2
william-a1.1965.0	login	1/31 22:51	R	50	medium	p5-a1-n11
william-a1.1960.0	login	1/31 17:13	I	50	hps_xlarge	

- destruction : `llcancel ref_job`
`llcancel william-a1.crihan.fr.2045`
`llcancel: Cancel command has been sent to the central manager.`

2.6. Soumission job séquentiel

```
# !/bin/csh
# Script de soumission Loadleveler, job sequentiel
# Nom du job
# @ job_name = job_sequentiel
# Nom des fichiers de sortie et d'erreur standard
# @ output = $(job_name).o$(jobid)
# @ error = $(job_name).e$(jobid)

# Type du job
# @ cri_job_type = serial
# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 1:00:00

# Mémoire maximale par processus (mb, gb, mw, gw, ..)
# @ data_limit = 600mb

# Stack maximale par processus (mb, gb, mw, gw, ..)
# @ stack_limit = 100mb
```

```
# Répertoire du compte utilisateur dont le contenu est copié
# @ cri_initialdir = /work/crihan/gm/JOB_SEQ/ENTREE

# Répertoire du compte utilisateur pour y placer les résultats
# @ cri_finadir = /work/crihan/gm/JOB_SEQ/SORTIE

# Politique d'envoi des mels
# @ notification = complete
# Adresse d'envoi des mels
# @ notify_user = gm@crihan.fr

# Obligatoire
# @ queue
```

```
###  
### Commandes utilisateur  
###  
# Déplacement dans le répertoire temporaire  
cd $LOCAL_WORK_DIR  
  
# Exécution du programme séquentiel  
if ($CRI_ARCH == pwr4) then  
  ./a.out_pwr4 > $LOCAL_SPOOL_DIR/OUT  
else  
  ./a.out_pwr5 > $LOCAL_SPOOL_DIR/OUT  
endif  
  
# Déplacement des fichiers à récupérer  
mv resultat.dat $LOCAL_SPOOL_DIR
```

2.7. Soumission job parallèle MPI

```
# !/bin/csh
# Script de soumission Loadleveler, job MPI
# Nom du job
# @ job_name = job_mpi

# Nom des fichiers de sortie et d'erreur standard
# @ output = $(job_name).o$(jobid)
# @ error = $(job_name).e$(jobid)

# Type du job
# @ cri_job_type = mpi
# Nombre de processus MPI
# @ blocking = 4 pour rester dans un seul nœud
# @ cri_total_tasks = 4

# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 1:00:00
```



```
# Mémoire maximale par processus (mb, gb, mw, gw, ..)
# @ data_limit = 1000mb

# Stack maximale par processus (mb, gb, mw, gw, ..)
# @ stack_limit = 120mb

# Répertoire du compte utilisateur dont le contenu est copié
# @ cri_initialdir = /work/crihan/gm/JOB_MPI/ENTREE
# Répertoire du compte utilisateur pour y placer les résultats
# @ cri_finaldir = /work/crihan/gm/JOB_MPI/SORTIE

# Politique d'envoi des mels
# @ notification = complete
# Adresse d'envoi des mels
# @ notify_user = gm@crihan.fr

# Obligatoire
# @ queue
```

```
###  
### Commandes utilisateur  
###  
# Déplacement dans le répertoire temporaire  
cd $LOCAL_WORK_DIR  
  
# Exécution du programme MPI  
if ($CRI_ARCH == pwr4) then  
  ./a.out_pwr4 > $LOCAL_SPOOL_DIR/OUT  
else  
  ./a.out_pwr5 > $LOCAL_SPOOL_DIR/OUT  
endif  
  
# Déplacement des fichiers à récupérer  
mv resultat.dat $LOCAL_SPOOL_DIR
```

2.8. Soumission job parallèle OpenMP

```
# !/bin/csh
# Script de soumission Loadleveler, job OpenMP
# Nom du job
# @ job_name = job_openmp

# Nom des fichiers de sortie et d'erreur standard
# @ output = $(job_name).o$(jobid)
# @ error = $(job_name).e$(jobid)

# Type du job
# @ cri_job_type = openmp
# Nombre maximal de threads OpenMP
# @ cri_total_tasks = 4

# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 1:00:00
```

```
# Mémoire maximale pour l'application (mb, gb, mw, gw, ..)
# @ data_limit = 1600mb

# Stack maximale pour l'application (mb, gb, mw, gw, ..)
# @ stack_limit = 400mb

# Répertoire du compte utilisateur dont le contenu est copié
# @ cri_initialdir = /work/crihan/gm/JOB_OMP/ENTREE
# Répertoire du compte utilisateur pour y placer les résultats
# @ cri_finaldir = /work/crihan/gm/JOB_OMP/SORTIE

# Politique d'envoi des mels
# @ notification = complete
# Adresse d'envoi des mels
# @ notify_user = gm@crihan.fr

# Obligatoire
# @ queue
```

```
###  
### Commandes utilisateur  
###  
# Déplacement dans le répertoire temporaire  
cd $LOCAL_WORK_DIR  
  
# Exécution du programme OpenMP  
if ($CRI_ARCH == pwr4) then  
  ./a.out_pwr4 > $LOCAL_SPOOL_DIR/OUT  
else  
  ./a.out_pwr5 > $LOCAL_SPOOL_DIR/OUT  
endif  
  
# Déplacement des fichiers à récupérer  
mv resultat.dat $LOCAL_SPOOL_DIR
```

2.9. Soumission job parallèle Gaussian

```
# !/bin/csh
# Script de soumission Loadleveler, job Gaussian
# Nom du job
# @ job_name = job_g03

# Nom des fichiers de sortie et d'erreur standard
# @ output = $(job_name).o$(jobid)
# @ error = $(job_name).e$(jobid)

# Type du job
# @ cri_job_type = gaussian
# Nombre maximal de threads Gaussian
# @ cri_total_tasks = 4

# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 1:00:00
```

```
# Mémoire maximale pour l'application (mb, gb, mw, gw, ..)
# @ data_limit = 2000mb

# Stack maximale pour l'application (mb, gb, mw, gw, ..)
# @ stack_limit = 400mb

# Répertoire du compte utilisateur dont le contenu est copié
# @ cri_initialdir = /work/crihan/gm/JOB_G03/ENTREE
# Répertoire du compte utilisateur pour y placer les résultats
# @ cri_finaldir = /work/crihan/gm/JOB_G03/SORTIE

# Politique d'envoi des mels
# @ notification = complete
# Adresse d'envoi des mels
# @ notify_user = gm@crihan.fr

# Obligatoire
# @ queue
```

```
###  
### Commandes utilisateur  
###  
# Déplacement dans le répertoire temporaire  
cd $LOCAL_WORK_DIR  
  
# Variables d'environnements nécessaires à Gaussian  
# N.B. : $LOCAL_WORK_DIR est dans /dlocal (partagé)  
# N.B. : $LOCAL_SCRATCH_DIR est dans /local (propre au noeud)  
  
setenv PATH "$${PATH}:::/soft/g03c02/g03"  
setenv g03root /soft/g03c02  
setenv GAUSS_SCRDIR $LOCAL_SCRATCH_DIR  
setenv GAUSS_EXEDIR /soft/g03c02/g03  
setenv LD_LIBRARY64_PATH /soft/g03c02/g03  
  
# Exécution du programme Gaussian  
g03 < gaussian.in > $LOCAL_SPOOL_DIR/gaussian.log  
# Déplacement des fichiers à récupérer  
mv gaussian.chk $LOCAL_SPOOL_DIR
```


3. Environnement de compilation XLF

3.1. Les commandes de compilation

Il existe une famille de compilateurs XLF selon le type d'application et le respect de la norme POSIX pour les threads.

⇒ Il est conseillé de toujours utiliser des compilateurs *threadsafes*, i.e. qui permettent l'exécution simultanée correcte d'une même portion de code (parallélisable !) par plusieurs threads :

⇒ suffixe `_r`

Type d'application	Fortran 77	Fortran 90	Fortran 95
Code séquentiel	<code>xlf_r</code>	<code>xlf90_r</code>	<code>xlf95_r</code>
Code parallèle MPI	<code>mpxlf_r</code>	<code>mpxlf90_r</code>	<code>mpxlf95_r</code>
Code parallèle OpenMP, PVM, P-threads, ...	<code>xlf_r</code>	<code>xlf90_r</code>	<code>xlf95_r</code>

L'édition des liens se fait avec la même commande que la compilation.

3.2. Options de compilation : entrées / sorties du compilateur

- **-qfixed=132** : fichier source au format fixe (-qfixed=72 défaut pour Fortran 77)
- **-qfree=f90** : fichier source au format libre (défaut pour Fortran 9x)
- **-I*dir*** : chemin pour les fichiers à inclure
- **-qsuffix=f=f90** (ou **f**) : extension des fichiers source
- **-qmoddir=*dir*** : précise le répertoire de création des fichiers modules

3.3. Options de compilation : portage

- **-qdpc=e** : les constantes réelles numériques sont converties en double précision
- **-qautodbl=dbl4** : conversion automatique des REAL(4) en REAL(8) et des COMPLEX(4) en COMPLEX(8)
- **-qnosave** : variables locales dynamiques et non pas statiques
- **-qundef** : rejet des déclarations implicites pour les variables
- **-qextname** : ajout d'un caractère _ à la fin de tous les noms d'objets venant d'autres systèmes où ils sont absents (exemple : flush)

3.4. Options de compilation : débogage

- **-qnooptimize** : pas d'optimisation du code
 - **-qcheck** : vérification des accès aux éléments de tableaux explicitement dimensionnés
 - **-qdbg** : informations pour le débogueur symbolique
 - **-qextchk** : vérification des informations sur les types de données dans les blocs communs, dans les définitions des procédures
 - **-qflttrap=ov:und:zero:inv:en** : type d'exceptions flottantes tracées
 - **-qfullpath** : inclusion du chemin absolu UNIX vers les fichiers sources
 - **-qinitauto=FF** : initialisation à NaN de toutes les variables automatiques
 - **-qfloat=nans** : détection des opérations flottantes utilisant des NaN
-
- Proposition d'options :
 - qnooptimize -qcheck -qdbg -qflttrap=ov:und:zero:inv:en -qfullpath
 - qinitauto=FF -qfloat=nans [-qextchk]

3.5. Options de compilation : profilage / optimisation

- **-O2, -O3** : niveau d'optimisation
- **-qstrict** : respect de la sémantique des programmes (si nécessaire)
- **-qhot** : optimisation plus agressives (**-O3** nécessaire)
- **-qunroll=[auto|yes]** : active ou renforce l'analyse des boucles
- **-qarch=pwr[5|4]** et **-qtune=pwr[5|4]** : optimisations spécifiques à l'architecture
- **-qipa** : analyse interprocédurale
- **-Q<x>** : inlining (**-Q**) ou non (**-Q!**), sélectif (**-Q+name1[:name2]** ou **-Q-name1[:name2]**)
→ nécessite **-qipa** et au moins **-O2**
- **-qsmp=omp** : parallélisation avec directives OpenMP
- **-pg** : prépare le programme au profilage par `gprof`
- **-qreport=[hotlist|smplist]** fichiers listing des transformations du code
- Proposition d'options :
-qarch=pwr[5|4] -qtune=pwr[5|4] [-qhot] -O3 [-qstrict] [-qsmp=omp] [-qipa]

3.6. Edition des liens

- **-bmaxdata:0x40000000** : 4 segments de 256 Mo pour les données du programme (*heap*)
- **-bmaxstack:0x10000000** : 256 Mo pour les variables locales (*stack*)

⇒ Options INutiles et pénalisantes en mode 64 bits

- **-brename:.name, .name_** : ajout d'un caractère souligné '_' à la fin d'un nom

⇒ Option utile pour le portage de `flush`, ...

- **-blpdata** : demande de larges pages (16 Mo) pour les applications consommatrices de mémoire

**Cette option est mise par défaut dans la configuration du compilateur
MAIS pour les binaires importés d'autres plate-formes AIX, il faut appliquer la com-
mande `ldedit` :**

`ldedit -blpdata <executable>`

- **-qipa** : si elle est présente lors de la compilation

4. Bibliothèques scientifiques

4.1. ESSL : Engineering and Scientific Subroutine Library

- **algèbre linéaire** : opérations sur des vecteurs et opérations matrice-vecteurs (BLAS 1 et 2)
- **calcul matriciel** (BLAS 3)
- **résolution de système linéaires** : divers types de stockage, divers types de matrice (sous-ensemble des routines BLAS niveau2 et niveau 3 ainsi que des routines LAPACK)
- **recherche de valeurs propres**
- **transformées de Fourier, convolutions, traitement du signal** : transformations de Fourier en 1D, 2D ou 3D
- **tris et recherche d'éléments** : recherches possibles pour des données de type entier, réel simple et réel double précision avec ou sans index
- **interpolation** : interpolation par polynômes, par splines cubiques en une et deux dimensions d'espace
- **quadrature numérique** : intervalles, finis, semi-infinis ou infinis
- **génération de nombres aléatoires** : loi uniforme et loi normale

4.2. P-ESSL : Parallel Engineering and Scientific Subroutine Library

- **PBLAS niveau 2** (ss-ens. des versions parallèles pour mémoire distribuée de BLAS 2)
- **PBLAS niveau 3** (ss-ens. des versions parallèles pour mémoire distribuée de BLAS 3)
- **résolution de système linéaires** : divers types de stockage, divers types de matrice (sous-ensemble des routines ScaLAPACK)
- **recherche de valeurs propres et valeurs singulières** (ss-ens. de routines ScaLAPACK)
- **transformées de Fourier** : transformations de Fourier en 2D ou 3D
- **génération de nombres aléatoires** : loi uniforme

Utilisation

Ajouter **-lessl** ou **-lpessl** lors de l'édition des liens

<http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp?topic=/com.ibm.cluster.essl.doc/esslbo>

4.3. LAPack : Linear Algebra Package

⇒ Bibliothèque d'algèbre linéaire : systèmes linéaires, ...

Utilisation

Ajouter **-lessl -llapack** lors de l'édition des liens

⇒ certaines routines LAPack sont dans ESSL sous forme optimisée

<http://www.netlib.org/lapack>

4.4. MASS : Mathematical Acceleration SubSystem

Bibliothèque mathématique optimisée

→ Fonctions exponentielle / logarithmique, trigonométriques, inverse, racine carrée, ...

⇒ Gain en performances **mais** pertes en arrondis.

Il existe une version pour traiter les vecteurs de données : **MASSv**

Utilisation

Ajouter **-lmass** et **-lmassv** lors de l'édition des liens

<http://techsupport.services.ibm.com/server/mass>



5. Outils

5.1. Déboguage

dbx et **pdbx** : Débogueurs symboliques standard

⇒ Compilation avec les options `-qnooptimize -qdbg -qfullpath`

5.2. Profilage

gprof : localisation de la consommation cpu

⇒ Compilation avec les options `-pg -qdbg -qfullpath + optimisations`

⇒ Exécution du programme séquentiel ou parallèle (MPI, OpenMP)

⇒ analyse du(des) fichier(s) `gmon.out`

⇒ `gprof a.out gmon.out > analyse_gprof`

⇒ `gprof a.out gmon.out.0 [gmon.out.1...] > analyse_gprof_all`

xprofiler : visualisation graphique plus conviviale que **gprof**

⇒ `xprofiler a.out gmon.out.0 [gmon.out.1...]`

Sortie de gprof : répartition du temps cpu

%	cumulative	self	self	self	total	name
time	seconds	seconds	calls	ms/call	ms/call	
36.9	92.23	92.23	87253	1.06	1.06	.saxpy [4]
29.9	167.06	74.83	45751	1.64	1.64	.pmv [5]
22.0	222.15	55.09	61502	0.90	0.90	.prodsca [6]
6.1	237.48	15.33	10000	1.53	3.17	.scdmb [7]
5.1	250.19	12.71	10000	1.27	21.85	.gradconj [3]
0.0	250.22	0.03				...mcount [8]
0.0	250.25	0.03				.qincrement [9]
0.0	250.27	0.02	1	20.00	20.00	.fctfx [10]
0.0	250.28	0.01	6	1.67	1.67	.nrmerr [11]
0.0	250.28	0.00	20000	0.00	0.00	.fctft [12]
0.0	250.28	0.00	252	0.00	0.00	..sigsetmask [13]
0.0	250.28	0.00	170	0.00	0.00	.pthread_mutex_lock [14]

⇒ **classement par ordre décroissant du cpu consommé.**

La répartition du nombre d'appels peut être trouvée en analysant la hiérarchie des appels de fonction.

Sortie de gprof : hiérarchie des appels

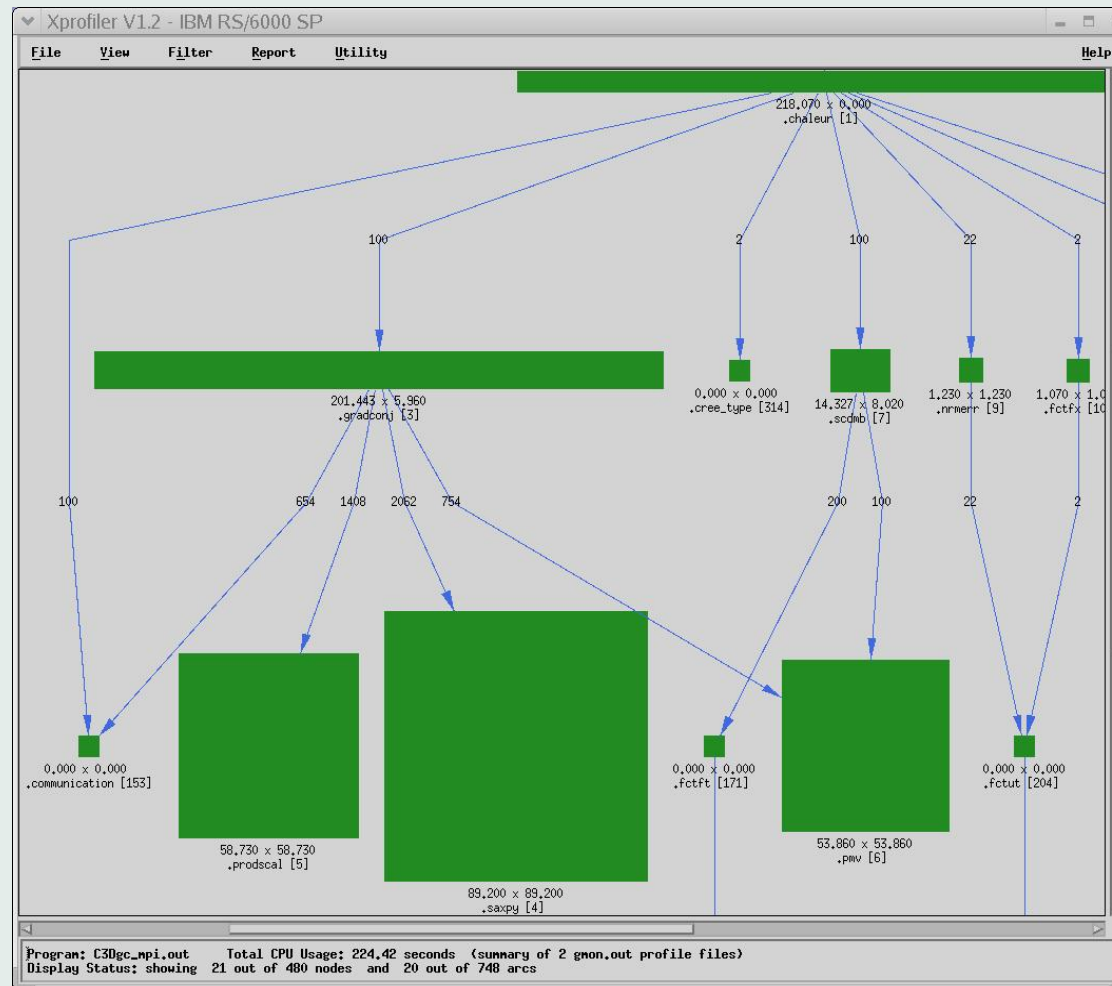
index	%time	self	descendants	called/total	parents	index
				called+self called/total	name	
		0.00	250.22	1/1	._start [2]	
[1]	100.0	0.00	250.22	1	.main [1]	
		12.71	205.79	10000/10000	.gradconj [3]	
		15.33	16.36	10000/10000	.scdmb [7]	
		0.02	0.00	1/1	.fctfx [10]	
		0.01	0.00	6/6	.nrmerr [11]	
		0.00	0.00	1/1	.domaine [88]	
		16.36	0.00	10000/45751	.scdmb [7]	
		58.47	0.00	35751/45751	.gradconj [3]	
[5]	29.9	74.83	0.00	45751	.pmv [5]	

fonctions parents : celles placées au-dessus dans le tableau

fonctions de référence : celles qui ont un numéro dans la colonne *index*

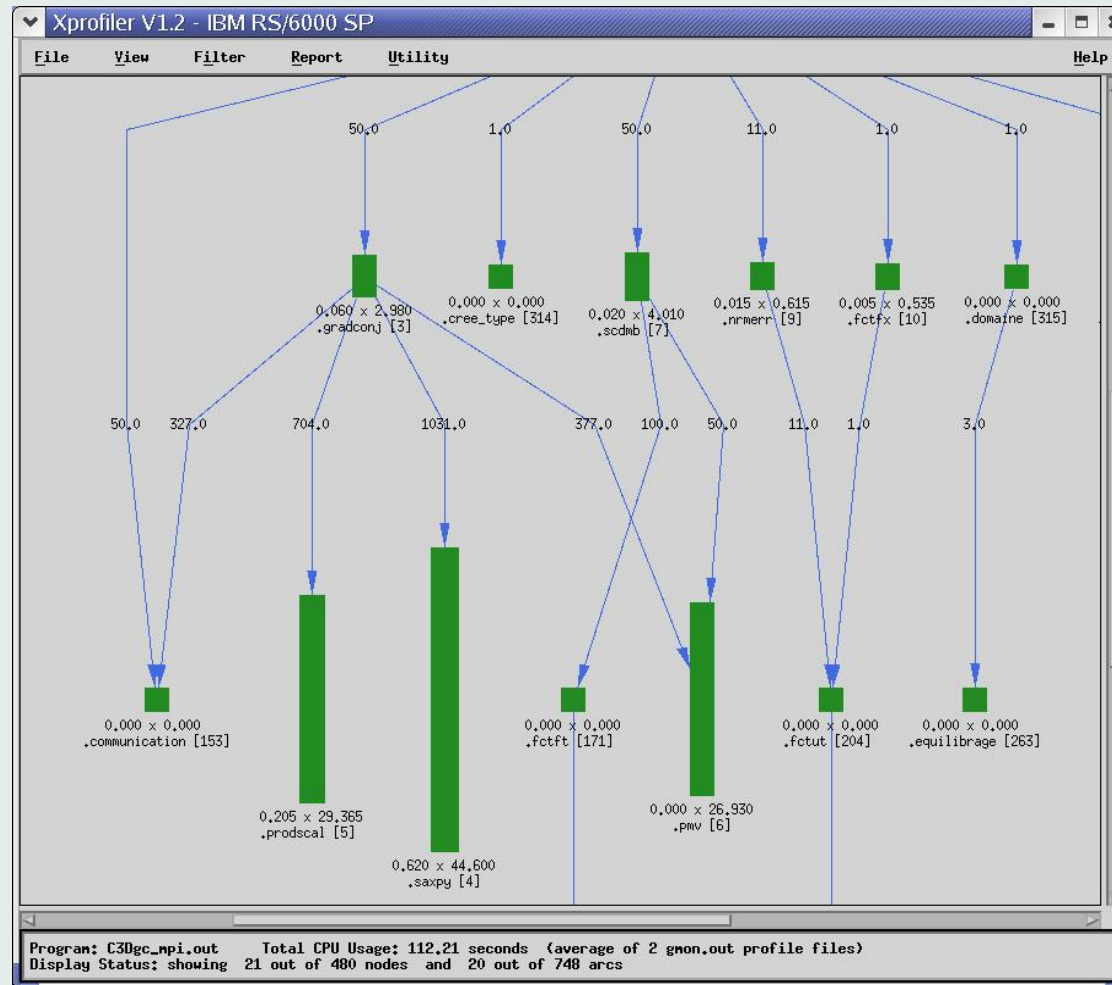
fonctions children : celles placées en-dessous dans le tableau

xprofiler mode summary : largeur (CPU total) x hauteur (CPU exclusif)



Chaque lien indique le nombre d'appels cumulé pour l'ensemble des processus

xprofiler mode average : largeur (écart-type procs.) x hauteur (moyenne des procs.)



Chaque lien indique le nombre d'appels moyen (nombre décimal) par processus

5.3. Collecte et visualisation : PE Benchmarker (outil IBM), jumpshot

- **Profile Collection Tool (pct)**

- traces d'évènements d'applications MPI (communications, ..) ou séquentielles ;
- profilage d'applications avec les compteurs matériels (cache misses, tlb misses, ...).

- utilitaires **ute** (Unified Trace Environment)

- utilitaires de conversion de fichiers de traces issus de **pct** pour exploitation avec **jumpshot** :
- **uteconvert** : conversion format traces AIX au format UTE ;
- **utestats** : statistiques sur fichiers au format UTE ;
- **utemerge** : fusion de fichiers au format UTE ;
- **slogmerge** : conversion fichiers UTE au format SLOG.

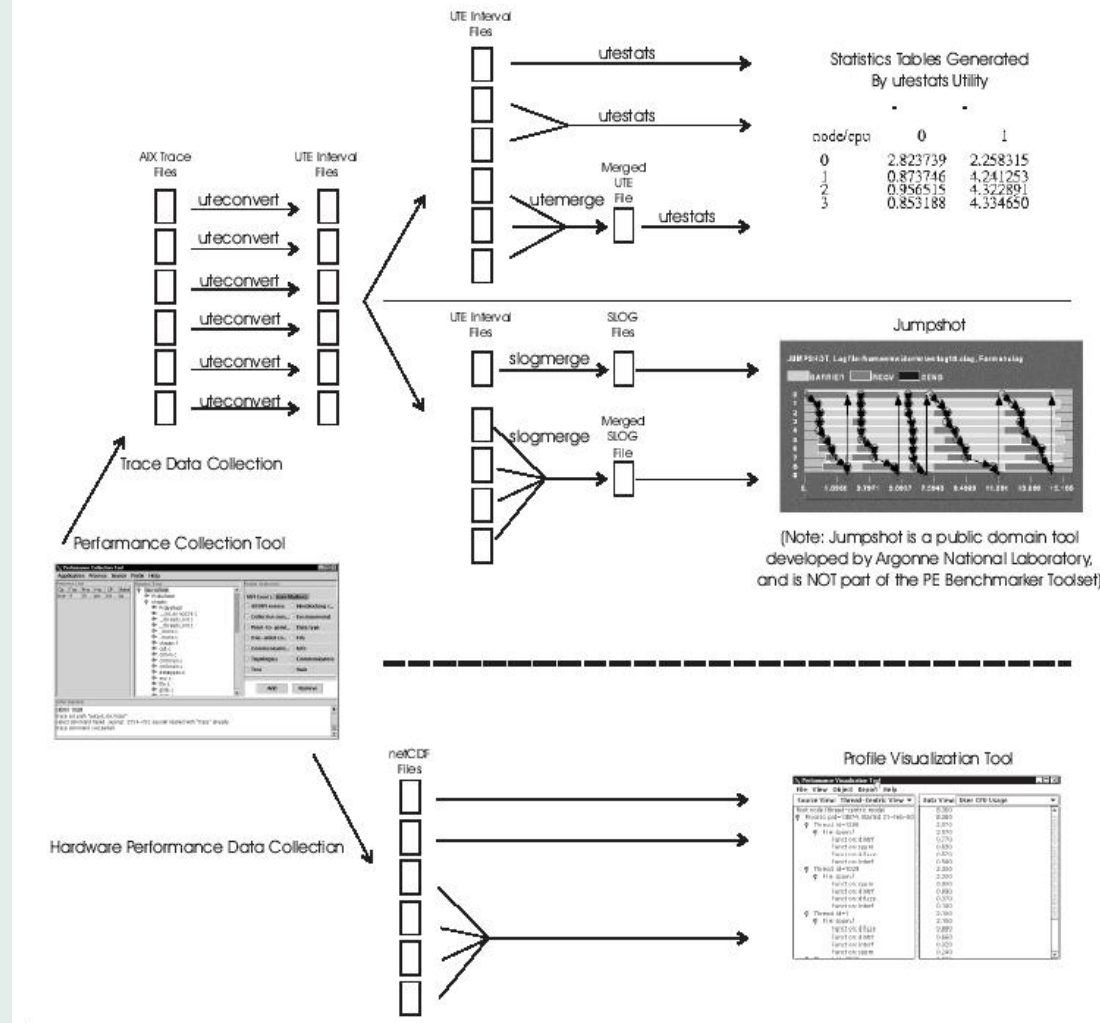
- **Profile Visualization Tool (pvt)**

- visualisation des fichiers de profilage (format netCDF) issus de **pct**

- **Jumpshot (Argonne National Laboratory)**

- logiciel domaine public de visualisation de traces d'évènements de fichiers au format SLOG

PE Benchmarker

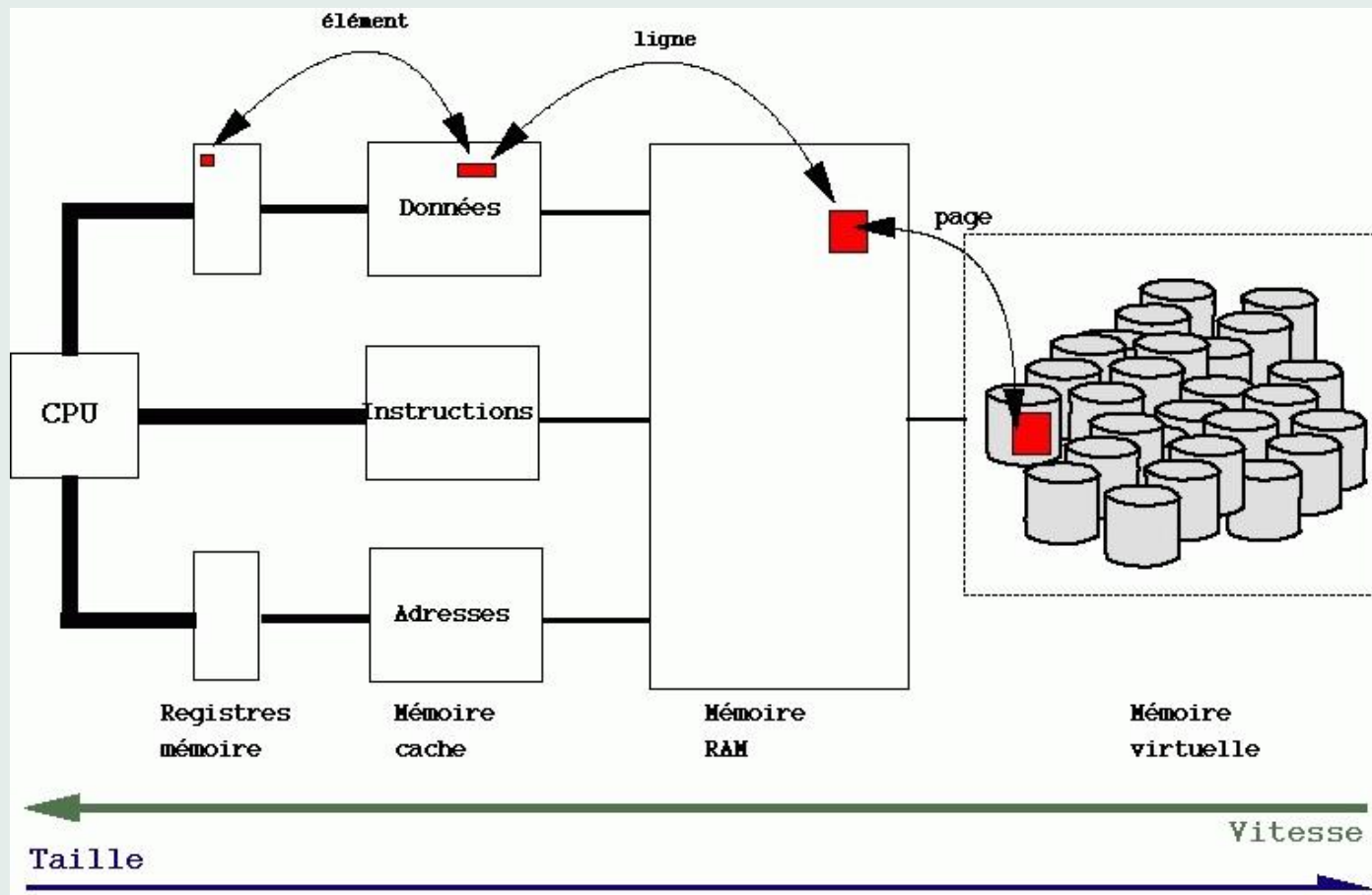


6. Considérations matérielles

- De nombreuses optimisations visent un parcours efficace de la mémoire
→ Important de comprendre son organisation
- Deux aspects critiques :
 - hiérarchie mémoire
 - localisations temporelle et spatiale
- Hiérarchie mémoire :
Structure mémoire similaire entre les architectures, basée sur une pyramide
 - au sommet : composants mémoire chers, rapides mais petits
 - à la base : composants mémoire moins chers, plus lents, plus gros
- Localisation spatiale et temporelle :
Caractéristiques du programme qui permettent d'augmenter les performances du programme

6.1. Hiérarchie mémoire

6.1.1. Parcours en mémoire

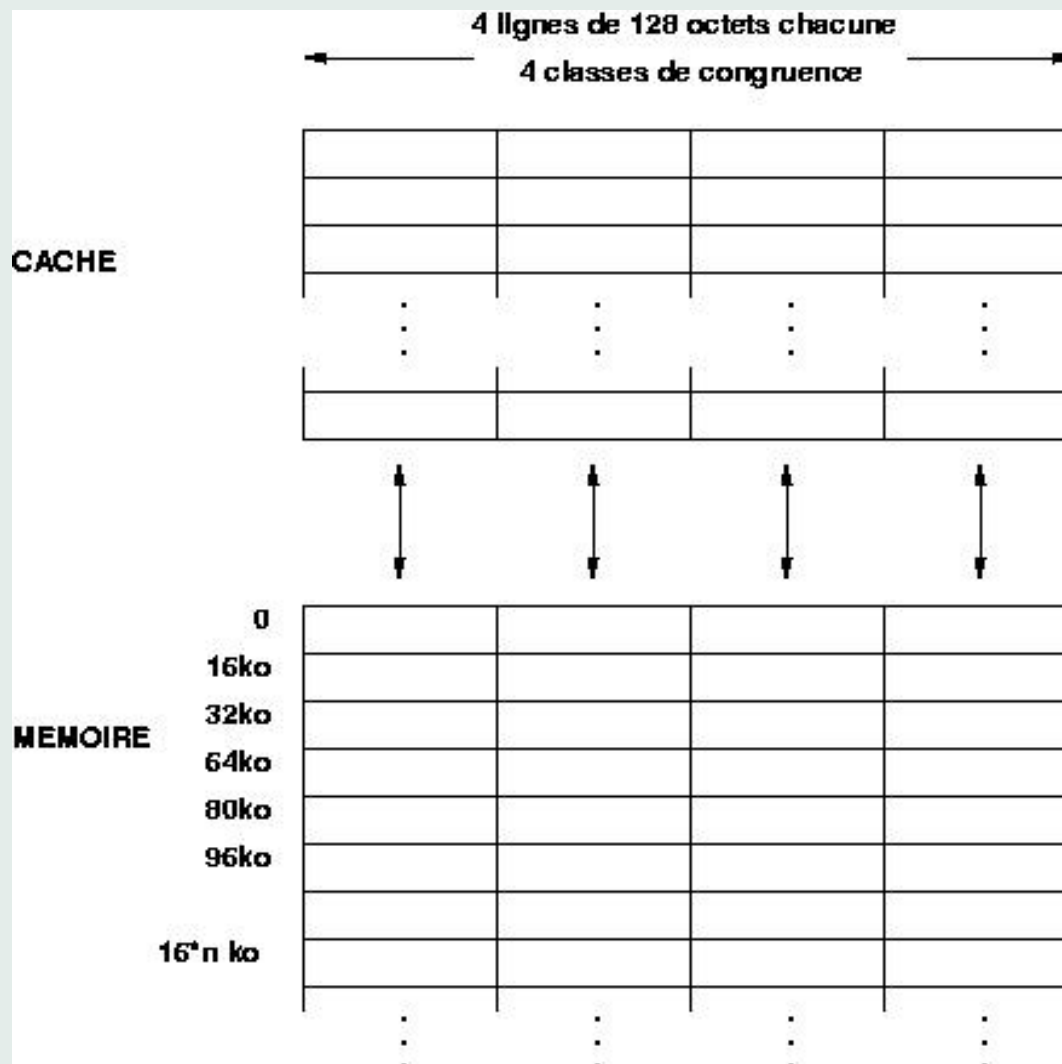


6.1.2. Latences (en cycles d'horloge) et associativité

	Power4	Power5
Registres	1	1
Cache L1	2	2
Cache L2	12	13
Cache L3	123	87
Mémoire	351	235

	Power4	Power5
Cache L1	2 voies associatives, FIFO 32 ko (D) + 64 ko (I) 128 octets	4 voies associatives LRU 32 ko (D) + 64 ko (I) 128 octets
Cache L2	4 voies associatives 1440 ko (par double cœur) 128 octets	10 voies associatives 1920 ko (un seul cœur) 128 octets
Cache L3	8 voies associatives 32 Mo 256 octets	12 voies associatives 36 Mo 256 octets

6.1.3. Associativité des caches



6.1.4. Défauts de caches et hors pages

- Quand une zone mémoire est référencée, ses lignes de cache associées sont vérifiées
 - Si la donnée s'y trouve, elle est chargée dans un registre
 - Autrement c'est un défaut de cache ou **cach miss**
 - Il faut alors la chercher au-delà, en mémoire
 - Cette perte de temps pénalise l'exécution du programme
- Rechercher une donnée signifie aussi parcourir une table rapide d'indexation des pages mémoire, la **Table Lookaside Buffer**
 - Elle contient les chemins vers les dernières pages mémoire accédées
 - Si la donnée n'est pas dans une de ses pages, il y a alors un défaut de TLB ou **TLB miss**
 - Il faut calculer l'adresse mémoire de la page, puis chercher la donnée
 - L'exécution du programme est pénalisée
- Si la donnée n'est pas en mémoire vive (**Page Fault**), il faut la chercher sur disque et le coût est prohibitif (> 1000 fois plus lent !).

6.2. Usage efficace de la mémoire

- Faire en sorte que la majeure partie des accès aient lieu dans la partie haute de la pyramide
Tenir compte de la localisation des données
 - Localisation spatiale : si une zone mémoire X est accédée, il est probable qu'une zone mémoire proche de X soit accédée prochainement
 - Localisation temporelle : si une zone mémoire X est accédée, il est probable qu'elle le sera à nouveau prochainement
- Tenir compte de ces localisations permet de réduire les défauts de cache, de TLB et de pages au travers de la ré-utilisation des données
- Réduire les défauts de cache
 - Eviter de parcourir les tableaux avec un pas (**stride**) grand ;
i.e. parcourir séquentiellement les données, selon leur représentation en mémoire
 - Eviter les conflits d'associativités de cache :
éviter les grandes puissances de 2 pour les premières dimensions des tableaux

- Réduire les défauts de TLB
 - contenance variable : souvent de l'ordre de 1024 pages mémoire référencées
 - Faire en sorte que si on parcourt plus de pages, l'essentiel du travail est fait
 - Eviter les `strides` trop grand : un `stride` = une entrée TLB

- Réduire les défauts de page
 - Déborder de la mémoire vive peut entraîner une forte dégradation des performances
 - Une solution : accéder à une machine avec plus de mémoire
 - Autrement : accéder aux données différemment par blocs restants en mémoire

- Utiliser au mieux le cache instructions (en général, pas de problème)

7. Optimisation scalaire

7.1. Ecriture du programme

- Réflexion au préalable sur papier
- Algorithmes performants (!)
- Déclaration explicite des variables, dimensions des tableaux, fonctions externes, ...
 - Directive `IMPLICIT NONE` dans tous les fichiers source,
 - Types génériques pour les variables (`REAL (KIND=)` en Fortran90, `INTEGER`)
 - Appels aux noms de fonctions génériques (`SQRT` au lieu de `DSQRT`)
 - Séparation des types dans les blocs `COMMON`
- Alignement des données dans les blocs `COMMON` (*padding*)
- Initialisation explicite des variables avant leur utilisation
- Commentaires dans le code source !

7.2. Bibliothèques scientifiques

⇒ **Ensembles de sous-programmes testés, validés et optimisés**

⇒ Bibliothèques disponibles sur le cluster IBM :

- BLAS 1, 2, 3
- FFT
- LAPack
- ESSL (contient BLAS 1, 2, 3 et une partie de LAPack)
- P-ESSL

⇒ **Edition des liens avec `-lessl -llapack [-lpessl]`**

- Bibliothèque mathématique optimisée : **MASS**

⇒ **Edition des liens avec `-lmass [-lmassv]`**

7.3. Modularité

- Fractionner le code source, commande `fsplit` (uniquement pour le Fortran 77)
- Mettre un sous-programme par fichier
- Mettre un bloc `COMMON` par fichier
- Contruire des bibliothèques thématiques
- Travailler avec un fichier de compilation, `Makefile`
- Tester les routines écrites sur des exemples

7.4. Validation du code

- Compilation du code en mode débogage
- Exécution d'un test représentatif

7.5. Analyse des performances

- Compilation du code avec des options d'optimisation
- Localisation de la consommation CPU à l'aide d'outils comme `gprof`

7.6. Optimisation par re-écriture

⇒ Travail très important car le compilateur ne fait pas tout, loin de là !

7.7. Opérations en virgule flottante

L'opération de base effectuée par le processeur Power 4 est : $\mathbf{A} + \mathbf{B} \times \mathbf{X}$

Une addition (soustraction) est faite avec $\mathbf{X} = \mathbf{1}$ ($\mathbf{X} = -\mathbf{1}$) ; une multiplication avec $\mathbf{A} = \mathbf{0}$

La division est traitée à part, par une structure matérielle spécifique (unique par processeur).

⇒ Dans une boucle, on peut remplacer plusieurs divisions par une multiplication par l'inverse ;

⇒ coût moins élevé, gains de performance, mais risque de perte de précision pour les arrondis.

Version non optimisée

```
DO i = 1, n
    A(i) = B(i) / C(i)
    P(i) = Q(i) / D(i)
END DO
```

Version optimisée

```
DO i = 1, n
    OCD = 1.0_rp / ( C(i) * D(i) )
    A(i) = B(i) * D(i)* OCD
    P(i) = Q(i) * C(i)* OCD
END DO
```

⇒ Calcul de puissances entières :

Version non optimisée	Version optimisée
<code>x = y **2.</code>	<code>x = y **2</code>
<code>ic = CMPLX (0., 1.)</code>	<code>ic = CMPLX (0.0_rp, 1.0_rp)</code>
<code>DO k = 1, n</code>	<code>ic2 = ic</code>
<code> A(k) = ic **k * B(k)</code>	<code>DO i = 1, n</code>
<code>END DO</code>	<code> A(k) = ic2 * B(k)</code>
	<code> ic2 = ic2 * ic</code>
	<code>END DO</code>

⇒ Tests arithmétiques flottants réalisés selon la précision machine :

```
IF ( ABS(a - b) < petit_machine_courante ) THEN ...
```

Remarque :

Eviter les puissances de 2 pour les dimensions des tableaux
 → cache thrashing

7.8. Arithmétique entière

- Ordre des opérations : $i * (i - 1) / 2$

$$i * \{(i - 1) / 2\} \neq \{i * (i - 1)\} / 2$$

⇒ Sans précaution, le niveau d'optimisation peut modifier le résultat

⇒ Ne pas se limiter à un niveau d'optimisation inférieur mais éliminer ces instructions dangereuses

- Evaluation des multiplications / divisions par puissances de 2

⇒ fonction **ISHFT** (entier, exposant signé)

⇒ $ir * 8 = \text{ISHFT}(ir, 3)$ NB : $2^3 = 8$

⇒ $(i * j * k) / 32 = \text{ISHFT}(i*j*k, -5)$ NB : $2^5 = 32$

7.9. Variables globales / locales

Utiliser des variables locales, automatiques, autant que possible

⇒ Analyse plus fine de l'utilisation des variables locales par le compilateur
(hors gros tableaux à allouer dynamiquement avec la commande **ALLOCATE**)

7.10. Expressions

Ecrire des expressions identiques en spécifiant les variables dans le même ordre

⇒ Le compilateur Fortran sait reconnaître des expressions identiques mais pas des permutations

⇒ $\mathbf{x = a + b + c + d}$ et $\mathbf{y = a + c + b + d}$ ne sont pas identiques ...

lorsque l'option **-qstrict** est présente

7.11. Boucles

- Taille des boucles raisonnable
- Eviter les expressions complexes pour les indices des tableaux
 - ⇒ calculs supplémentaires pour l'adresse des éléments
 - ⇒ perte de performances dans le chargement des données
- Eviter des types de données de taille intermédiaire : `INTEGER*1`, `REAL*16`, ...

- Sortir des boucles les tests conditionnels invariants

Version non optimisée

```
DO i = 1, n
  IF ( D(j) < 0 ) X(i) = 0.0_rp
  A(i) = B(i) + C(i) * D(i)
  E(i) = X(i) + F * G(i)
END DO
```

Version optimisée

```
IF ( D(j) < 0 ) THEN
  DO i = 1, n
    A(i) = B(i) + C(i) * D(i)
    X(i) = 0.0_rp
    E(i) = F * G(i)
  END DO
ELSE
  DO i = 1, n
    A(i) = B(i) + C(i) * D(i)
    E(i) = X(i) + F * G(i)
  END DO
END IF
```

- Traiter les premières / dernières itérations à part

Version non optimisée

```

DO i = 1, n
  IF ( i == 1 ) THEN
    X(i) = 0.0_rp
  ELSE IF ( i == n )
    X(i) = 1.0_rp
  END IF
  A(i) = B(i) + C(i) * D(i)
  E(i) = X(i) + F * G(i)
END DO

```

Version optimisée

```

A(1) = B(1) + C(1) * D(1)
X(1) = 0.0_rp
E(1) = X(1) + F * G(1)
DO i = 2, n-1
  A(i) = B(i) + C(i) * D(i)
  E(i) = X(i) + F * G(i)
END DO
X(n) = 1.0_rp
A(n) = B(n) + C(n) * D(n)
E(n) = X(n) + F * G(n)

```

Alternative :

utiliser des scalaires de la forme

$$iq = 1/i = \begin{cases} 1, & \text{si } i = 1 \\ 0, & \text{si } i > 1 \end{cases}$$

Attention au coût de la division entière !

- La fusion de boucles consiste à regrouper les instructions de plusieurs boucles en une seule

⇒ Avantages :

→ meilleur recouvrement des instructions

→ réduction des défauts de cache si les mêmes tableaux sont traités

→ réduction du coût de gestion de boucle (avantage mineur) ⇒ Inconvénients :

→ risque d'accroissement des défauts de cache en raison de l'associativité du cache

→ accès à des tableaux qui occupent une plus grande part du cache

Version non optimisée

```
DO I=1, ARRAY_SIZE
  X = X * A(I) + B(I)
END DO
DO I=1, ARRAY_SIZE
  Y = Y * A(I) + C(I)
END DO
```

Version optimisée

```
DO I=1, ARRAY_SIZE
  X = X * A(I) + B(I)
  Y = Y * A(I) + C(I)
END DO
```


- Sortir des boucles les appels aux sous-programmes :
⇒ faire de l'*inlining* ou transmettre le tableau comme argument au sous-programme
- Limiter le nombre d'appels aux fonctions intrinsèques
⇒ N appels à la fonction SIN au lieu de N^2 appels

Version non optimisée

```
DO i = 1, n
  DO j = 1, n
    A(j,i) = B(j,i) * SIN( X(j) )
  END DO
END DO
```

Version optimisée

```
DO j = 1, n
  SINX(j) = SIN( X(j) )
END DO
DO i = 1, n
  DO j = 1, n
    A(j,i) = B(j,i) * SINX(j)
  END DO
END DO
```

- Utiliser des compteurs de boucles de type INTEGER (INTEGER*8 en mode 64 bits)
- Eviter les constructions utilisant des GOTO, ASSIGN, STOP, PAUSE

- Accès séquentiel des données : *stride* (pas) de 1 si possible
⇒ Attention aux différences de stockage Fortran / C

++	+	-	--
<pre>DO j = 1, M DO i = 1, N A(i,j) END DO END DO</pre>	<pre>DO j = 1, M DO i = 1, N A(i+(j-1)*N) END DO END DO</pre>	<pre>DO j = 1, M DO i = 1, N k = k + 1 A(k) END DO END DO</pre>	<pre>DO j = 1, M DO i = 1, N A(index(i,j)) END DO END DO</pre>

- réduire les boucles extérieures
- fusionner les boucles consécutives ayant les mêmes bornes

- Augmenter le débit entre la mémoire et les registres du processeur
⇒ chaque processeur dispose de 8 **streams** qui lui permettent de créer des flux de données.

La boucle suivante ...

```
REAL*8  A1(N) , A2(N) , A3(N) , A4(N) , A5(N)
REAL*8  B1(N) , B2(N) , B3(N) , B4(N) , B5(N)
REAL*8  C1(N) , C2(N) , C3(N) , C4(N) , C5(N)
```

```
DO i = 1, N
  C1(i) = A1(i) * B1(i)
  C2(i) = A2(i) * B2(i)
  C3(i) = A3(i) * B3(i)
  C4(i) = A4(i) * B4(i)
  C5(i) = A5(i) * B5(i)
END DO
```

- ⇒ 3 flux de données au lieu de 15.
- ⇒ 3 lignes de cache L1 au lieu de 15 par itération.
- ⇒ Les données cachées sont utilisées en 4 itérations au lieu de 16.

Si plus de 8 flux sont nécessaires, fractionnez la boucle !

... peut être remplacée par ...

```
REAL*8  A(5,N)
REAL*8  B(5,N)
REAL*8  C(5,N)
```

```
DO i = 1, N
  C(1,i) = A(1,i) * B(1,i)
  C(2,i) = A(2,i) * B(2,i)
  C(3,i) = A(3,i) * B(3,i)
  C(4,i) = A(4,i) * B(4,i)
  C(5,i) = A(5,i) * B(5,i)
END DO
```

- Déroulage des boucles (*unrolling*)

- * Effectuer plusieurs itérations d'une même boucle simultanément

<pre>DO i = 1, n, 1 ... (i) ... END DO</pre>	<pre>DO i = 1, n, unroll ... (i) (i+1) (i+2) (i+unroll-1) ...</pre>
--	---

- * Avantages :

- plus d'opportunités pour le processeur
- plus de ré-utilisation de données
- exploitation des lignes de cache
- réduction du coût d'indice de boucles (avantage mineur)

- * Inconvénients :

- plus de registres nécessaires
- code supplémentaire pour traiter les itérations restantes

(do i=1,n-mod(n,unroll),unroll & do i=n-mod(n,unroll)+1,n)

N B : Déroulement boucle externe : augmente le ratio opérations / (load & store)
Déroulement boucle interne : augmente la masse d'opérations indépendantes par itération

Unrolling : deux stratégies

1. Dérouler une boucle de manière séquentielle

```
DO i = 1, n, p
  ... (i) ...
  ... (i+1) ...
  ... (i+k) ...
  ... (i+p-1) ...
END DO
```

2. Dérouler une boucle de manière discrète

```
DO i = 1, n/p, 1
  ... (i) ...
  ... (i+n/p) ...
  ... (i+k*n/p) ...
  ... (i+(p-1)*n/p) ...
END DO
```

- Tenir compte du nombre de références mémoire (hors scalaires) par itération :
→ éviter de saturer le cache en déroulant trop
- Tenir compte des spécificités architecturales des processeurs :
→ *Data Prefetch Streams* pour la famille des processeurs PowerPC
→ pas plus de 8 références mémoire dans ce cas

Exemple :

```

DO i = 1, n
  DO j = 1, n
    y (i) = y(i) + x(j)*a(j,i)
  END DO
END DO

```

Avant : 8 loads pour 4 itérations

Après : 5 loads pour 1 itération déroulée 4 fois

4 FMA (*FMA = Floating point Multiply and Add*)

Ratio : 8 loads / 4 FMA \mapsto 5 loads / 4 FMA

s0, s1, s2, s3 :

scalaires temporaires importants pour limiter les

MàJ intempestives de $y(i), \dots, y(i+3)$

```

DO i = 1, n, 4
  s0 = y(i)
  s1 = y(i+1)
  s2 = y(i+2)
  s3 = y(i+3)
  DO j = 1, n
    s0 = s0 + x(j)*a(j,i)
    s1 = s1 + x(j)*a(j,i+1)
    s2 = s2 + x(j)*a(j,i+2)
    s3 = s3 + x(j)*a(j,i+3)
  END DO
  y(i) = s0
  y(i+1) = s1
  y(i+2) = s2
  y(i+3) = s3
END DO

```

8. Calcul parallèle : Quelques définitions

8.1. Qu'est-ce que le calcul parallèle ?

Le calcul parallèle est un ensemble de techniques **matérielles** et **logicielles** qui permettent l'exécution **simultanée** de séquences d'instructions **indépendantes** sur plusieurs processeurs.

Techniques matérielles : processeurs, mémoire, réseaux d'intercommunication, ...

Techniques logicielles : compilateurs, langage parallèle, bibliothèques, de passage de messages ou scientifiques, ...

8.2. Pourquoi faire du calcul parallèle ?

Le calcul parallèle a plusieurs avantages :

- il permet d'obtenir des **temps de restitution plus courts** en distribuant le travail à effectuer ;
- il permet d'effectuer des **calculs plus gros** en morcelant l'application sur plusieurs processeurs, voire nœuds, voire calculateurs et ainsi elle dispose de plus de ressources matérielles (notamment la ressource mémoire).

8.3. Accélération et Efficacité

Notation : $T(p)$ = temps d'exécution sur p processus

Accélération (Speedup) : $A(p) = T(1) / T(p)$

Efficacité (Efficiency) : $E(p) = A(p) / p$

On prend comme référence le temps du meilleur algorithme séquentiel.

8.4. Loi d'Amdhal

⇒ Gain possible de la parallélisation d'une application séquentielle :

$$A(par, seq) = 1 / (seq + par / N)$$

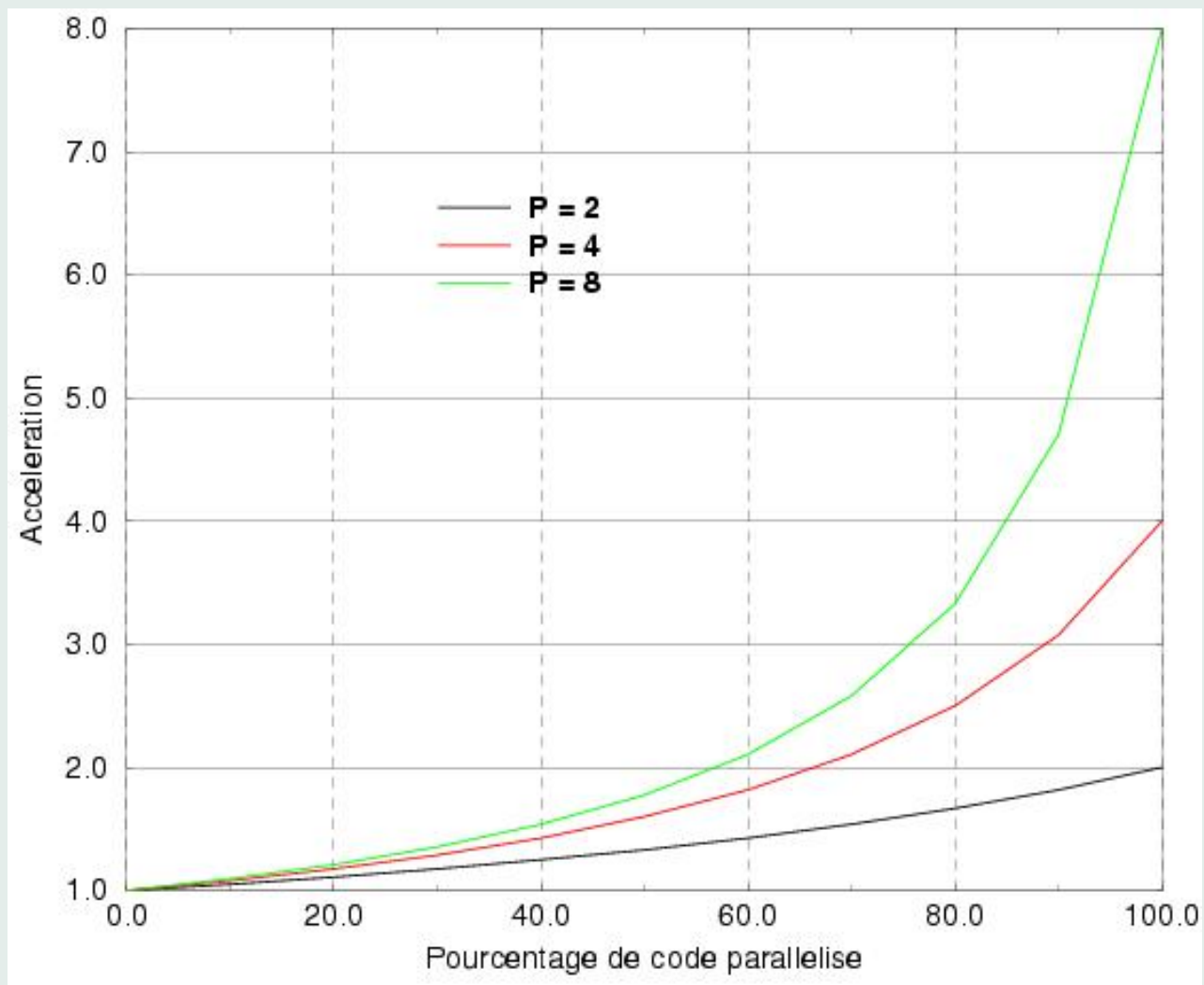
$$\left. \begin{array}{l} seq : \text{portion séquentielle du code} \\ par : \text{portion parallèle du code} \end{array} \right\} par + seq = 1$$

N : nombre de processeurs

La part séquentielle de l'application est une borne supérieure pour l'accélération.

Exemple : $seq = 0.10$ (10%) alors $par = 0.90$ et $A(par, seq) < 1/seq = 10 \quad \forall N$

Loi d'Amdahl



9. Parallélisme par passage de messages avec MPI

- Il repose sur l'**échange de messages** entre les processus pour le transfert de données, les synchronisations, les opérations globales
- **La gestion de ces échanges est réalisée par MPI** (Message Passing Interface)
- Cet ensemble repose sur le principe du **SPMD** (*Single Program Multiple Data*)
- Chaque processus dispose de ses **propres** données, sans accès direct à celles des autres
- **Explicite**, cette technique est entièrement à la charge du développeur
- Ces échanges qui impliquent deux ou plusieurs processus se font dans un **communicateur**
- Chaque processus est identifié par son **rang**, au sein du groupe

9.1. Environnement

- Initialisation en début ([MPI_INIT](#))
- Finalisation en fin de programme ([MPI_FINALIZE](#))

```
INTEGER :: nbprocs, myrank
INTEGER :: ierr = 0
!
CALL MPI_INIT ( ierr )
!
CALL MPI_COMM_SIZE ( MPI_COMM_WORLD, nbprocs, ierr )
CALL MPI_COMM_RANK ( MPI_COMM_WORLD, myrank, ierr )
!
CALL MPI_FINALIZE ( ierr )
```

- Pour réaliser des opérations impliquant des données d'autres processus, il est nécessaire d'échanger ces informations aux travers de [messages](#)
- Ces messages se font sous la forme de [communications](#) impliquant au moins deux processus
- On peut faire une analogie avec le courrier électronique

9.2. Structures de données

- Les données transmises sont typées
- Types prédéfinis : `MPI_INTEGER`, `MPI_REAL`, ...
- Type homogène :
 - données contiguës : `MPI_TYPE_CONTIGUOUS`
⇒ colonne de matrice en Fortran
 - données distantes d'un pas constant : `MPI_TYPE_VECTOR` ou `MPI_TYPE_HVECTOR`
⇒ ligne ou bloc d'une matrice
 - données distantes d'un pas variable : `MPI_TYPE_INDEXED` ou `MPI_TYPE_HINDEXED`
⇒ triangle dans une matrice
- Type hétérogène :
⇒ Construction d'une structure: `MPI_TYPE_STRUCT`
- Validation d'un type : `MPI_TYPE_COMMIT`
- Destruction d'un type : `MPI_TYPE_FREE`

9.3. Communications point à point

- La communication point à point est une **communication entre deux processus** :
⇒ expéditeur et destinataire
- Composition d'un message :
 - le communicateur (**comm**)
 - les deux identifiants (**src** et **dest**)
 - la donnée (**buf**), son type (**datatype**) et sa taille (**count**)
 - une étiquette (**tag**) qui permet au programme de distinguer différents messages

Communications synchrones et asynchrones :

```
CALL MPI_SEND ( buf, count, datatype, dest, tag, comm, ierr )
```

```
CALL MPI_RECV ( buf, count, datatype, src , tag, comm, ierr )
```

```
CALL MPI_ISEND( buf, count, datatype, dest, tag, comm, irq, ierr )
```

```
CALL MPI_IRECV( buf, count, datatype, src , tag, comm, irq, ierr )
```

un identifiant (**irq**) de la requête pour les messages asynchrones

9.4. Communications collectives

- La communication collective est une **communication qui implique un ensemble de processus qui l'effectuent tous**
- Il y a plusieurs types de communication collective :

9.4.1. les synchronisations globales

c'est une barrière de synchronisation qui agit sur l'ensemble des membres d'un communicateur.

```
CALL MPI_BARRIER ( comm, ierr )
```

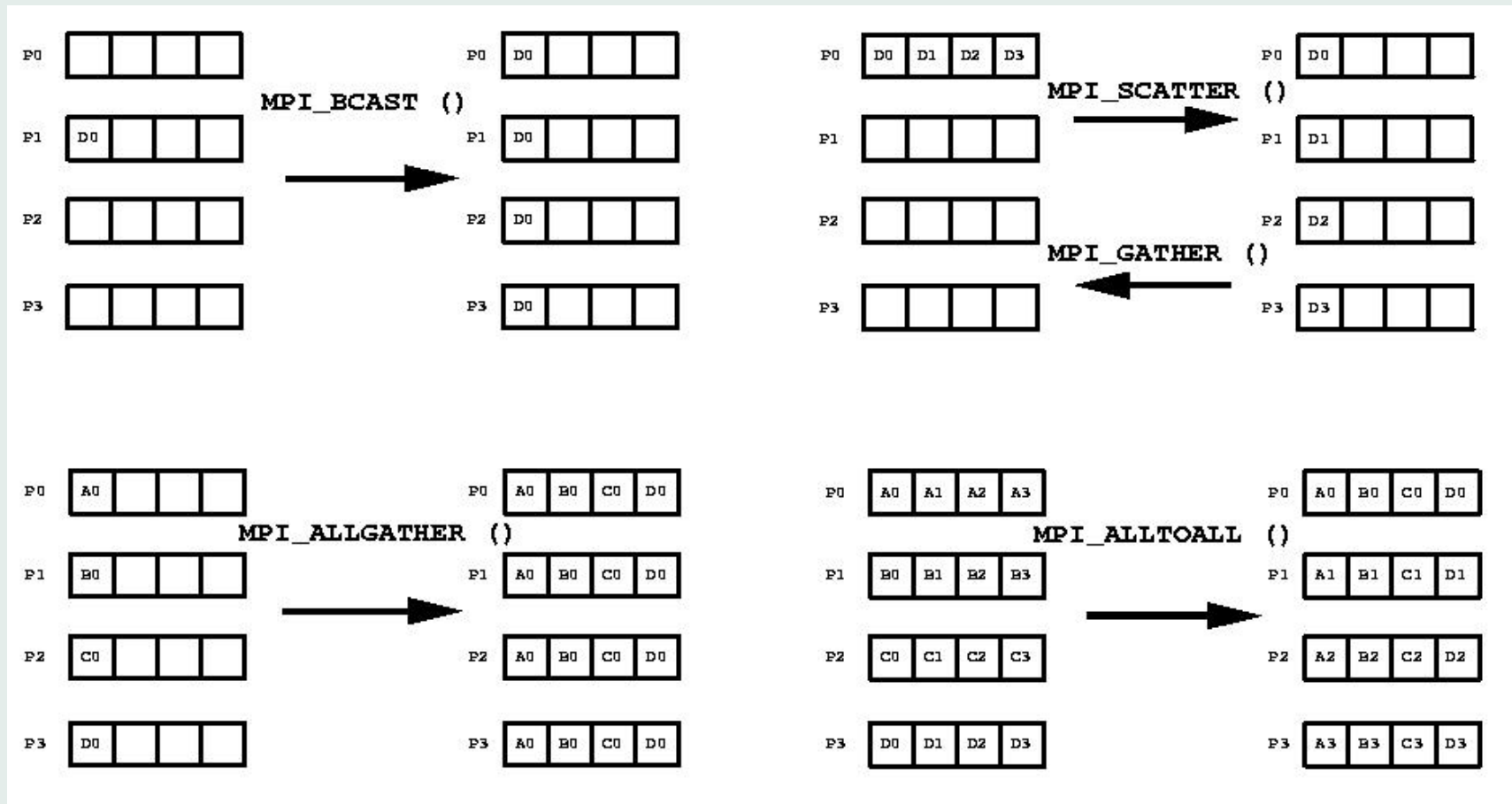
9.4.2. les opérations de réduction sur des données réparties

somme, produit, maximum ... effectué sur des données réparties (**MPI_REDUCE**) et le résultat peut être ensuite redistribué (**MPI_ALLREDUCE**).

```
CALL MPI_REDUCE ( sbuf, rbuf, count, datatype, oper, root, comm, ierr )
```

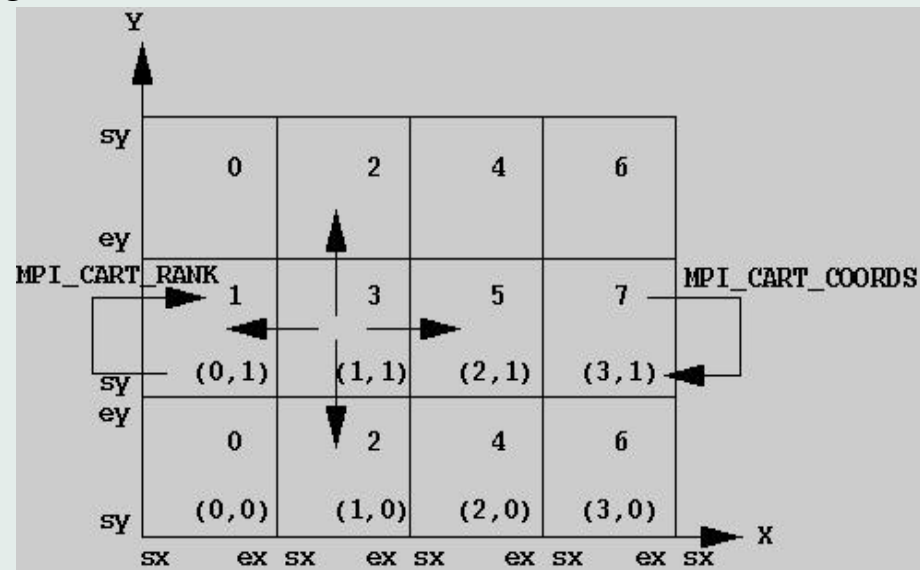
```
CALL MPI_ALLREDUCE ( sbuf, rbuf, count, datatype, oper, comm, ierr )
```

9.4.3. Diffusion / collecte



9.5. Topologie

- Topologie cartésienne : grille de processus
- Nombre de processus par dimension d'espace : `MPI_DIMS_CREATE`
- Création de la grille : `MPI_CART_CREATE`
 - ⇒ périodicité ou non des conditions aux limites
 - ⇒ création d'un nouveau communicateur
- Recherche des voisins dans chaque dimension : `MPI_CART_SHIFT`
- Coordonnées / rang : `MPI_CART_COORDS` et `MPI_CART_RANK`



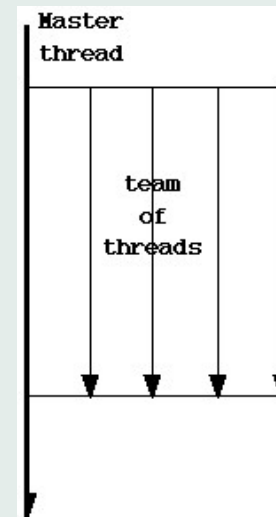
9.6. Utilisation de MPI sur le cluster IBM

- Compilation : `mpxlf_r`, `mpxlf90_r`, `mpxlf95_r`
- Edition des liens : rien de spécial ! (pas de `-lmpi`)
- Commande de lancement :
 - ⇒ `poe ./a_mpi.out -procs N`
 - ⇒ `a_mpi.out -procs N`
- Soumission job :
 - `cri_job_type` → `mpi`
 - `cri_total_tasks` → `N`
- Exemple récapitulatif : équation de la chaleur en 3D par décomposition de domaine
http://www.crihan.fr/calcul/tech/doc_ibm_pwr5/C3d

10. Parallélisme à mémoire partagée avec OpenMP

OpenMP est un ensemble de **constructions parallèles** basées sur des **directives** de compilation pour architecture à **mémoire partagée**.

- Il est basé sur le principe du **fork and join**
- Une équipe de threads est créée à l'entrée d'une région parallèle
- Son effectif est contrôlé par variable d'environnement ou appel librairie
- Avant et après, l'exécution est séquentielle
- Les threads sont identifiées par leur **rang** et celui de la thread master est 0



10.1. Directives

La parallélisation se fait en insérant des **directives** dans le code séquentiel :

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

```
nbth = 1
```

```
!$      nbth = OMP_GET_NUM_THREADS( )
```

Où :

!\$OMP ou C\$OMP

est une sentinelle, obligatoire comme préfixe des directives

!\$ ou C\$

permet d'effectuer une compilation conditionnelle
si OpenMP est reconnu

PARALLEL

est une directive

[clause [,clause] ...]

sont les clauses, optionnelles,
qui permettent de décrire la visibilité des variables

10.2. Variables privées

Les variables `PRIVATE` et `FIRSTPRIVATE` ont une adresse unique dans chaque thread parallèle : elles sont dupliquées.

Elles ne sont pas accessibles en dehors de leur région parallèle.

Seules, celles déclarées comme `FIRSTPRIVATE` sont initialisées au début de la région.

Les constantes (variables avec l'attribut `PARAMETER`) et les arguments en lecture (attribut `INTENT (IN)`) ne peuvent pas être privés.

Les variables privées qui conservent leur valeur d'une région parallèle à une autre sont appelées `THREADPRIVATE`.

Les variables `THREADPRIVATE` peuvent être initialisées par la thread master au travers de la clause `COPYIN`.

Exemples : indices de boucle (par défaut), variables scalaires (résultats intermédiaires), variables locales, tableaux automatiques, ...

10.3. Variables partagées

Par défaut dans les régions parallèles, toutes les variables sont partagées (ou publiques).

La déclaration se fait avec la clause `SHARED`.

Toutes les threads accèdent à la même instance de la variable (**attention aux conflits !**).

Exemples : variables locales rémanentes (`DATA`, `SAVE`), celles déclarées dans des `commons` ou `modules`, ou encore passées en argument sont partagées.

10.4. Variables de réduction

Une clause de réduction `REDUCTION(op:variable)` permet d'effectuer des réductions sur des variables scalaires partagées (`variable`) avec des opérateurs associatifs (`op`) ; extension aux tableaux en OpenMP 2.0

Cette `variable` doit être `SHARED` avant le début de la construction parallèle.

Exemples : produit scalaire, maximum, somme, ET logique, ...

10.5. Constructions work sharing

Elles permettent de partager le travail entre les threads parallèles automatiquement.

Elles sont incluses dans les régions parallèles, n'ont pas de synchronisation en entrée et la clause `NOWAIT` permet de lever celle en sortie.

```
SINGLE [clause[,clause...]] ... END SINGLE [nowait]
```

```
SECTIONS [clause[,clause...]] ... SECTION ... END SECTIONS[nowait]
```

```
DO [clause[,clause...]] ... END DO [nowait]
```

Une région parallèle peut se limiter à une construction work sharing.

```
PARALLEL SECTIONS [clause[,clause...]] ... SECTION ...  
END PARALLEL SECTIONS [nowait]
```

```
PARALLEL DO [clause[,clause...]] ... END PARALLEL DO [nowait]
```

Boucle dans une région parallèle :

```
!$OMP DO
    DO i = 1, n
        CALL mywork(i)
    END DO
!$OMP END DO
```

Construction parallèle restreinte à une triple boucle :

```
!$OMP PARALLEL DO DEFAULT(NONE) &
!$OMP SHARED (x, y, z, nx, ny, nz) &
!$OMP PRIVATE (i, j, k)
    DO k = 1, nz
        DO j = 1, ny
            DO i = 1, nx
                x(i,j,k) = y(i,j,k) * z(i,j,k)
            END DO
        END DO
    END DO
!$OMP END PARALLEL DO
```

10.6. Sérialisation et synchronisations

Il y a plusieurs constructions qui permettent de spécifier l'ordre d'accès à des données partagées

directive <code>MASTER ... END MASTER</code>	accès pour la thread de rang 0 uniquement
section <code>CRITICAL ... END CRITICAL</code>	accès pour une seule thread à la fois
directive <code>ATOMIC</code>	section critique formée d'une seule instruction
barrière <code>BARRIER</code>	barrière de synchronisation globale

Remarques :

La directive `MASTER ... END MASTER` n'a pas de synchronisation implicite à la fin contrairement à `SINGLE ... END SINGLE`

La directive `MASTER ... END MASTER` est moins chère que la section `SINGLE ... END SINGLE`

La directive `ATOMIC` est moins chère que la section `CRITICAL ... END CRITICAL`

10.7. Utilisation d'OpenMP sur le cluster IBM

- Compilation : `xlf_r`, `xlf90_r`, `xlf95_r` et option `-qsmp=omp`
compilateurs threadsafes obligatoirement !
- Edition des liens : même compilateur et option `-qsmp=omp`
- Commande de lancement :
 - ⇒ `setenv OMP_NUM_THREADS N`
 - ⇒ `./a_omp.out`
- Soumission job :
 - `cri_job_type` → `openmp`
 - `cri_total_tasks` → `N`
- Exemple récapitulatif : équation de la chaleur en 3D par partage du travail
http://www.crihan.fr/calcul/tech/doc_ibm_pwr5/C3d

11. Exemple récapitulatif : équation de la Chaleur 3D

11.1. définition du problème

- Les équations ...

$$\begin{cases} \frac{\partial u}{\partial t} - \lambda \Delta u = f & \text{dans } \Omega, \\ u = 0 & \text{sur } \Gamma = \partial\Omega \end{cases} \quad \text{avec } \Omega = [0, 1]^3$$

- La parallélisation est faite :
par décomposition de domaine avec MPI
par partage du travail avec OpenMP.
- La discrétisation spatiale est faite par un schéma aux différences finies d'ordre deux à trois points
⇒ maillage uniforme noté x_i, j, k
⇒ pas de discrétisation hx, hy, hz respectivement selon les directions spatiales.
- La discrétisation temporelle est faite par le schéma de Crank-Nicholson, ordre 2 semi-implicite et inconditionnellement stable dans notre cas, avec dt le pas de la discrétisation.

$$\{I - 0.5\lambda dt \Delta\} u^{n+1} = \{I + 0.5\lambda dt \Delta\} u^n + 0.5dt \{f^{n+1} + f^n\}$$

- Le laplacien est ainsi discrétisé :

$$\Delta_h u(i, j, k) = \frac{u_{i-1,j,k} - 2.u_{i,j,k} + u_{i+1,j,k}}{h_x^2} + \frac{u_{i,j-1,k} - 2.u_{i,j,k} + u_{i,j+1,k}}{h_y^2} + \frac{u_{i,j,k-1} - 2.u_{i,j,k} + u_{i,j,k+1}}{h_z^2} \quad (\text{termes d'erreurs en } h_x^2, h_y^2, h_z^2).$$

- Le système global s'écrit alors sous la forme :

$$u_{i,j,k}^{n+1} - 0.5\lambda dt \left\{ \frac{u_{i-1,j,k}^{n+1} - 2u_{i,j,k}^{n+1} + u_{i+1,j,k}^{n+1}}{h_x^2} + \frac{u_{i,j-1,k}^{n+1} - 2u_{i,j,k}^{n+1} + u_{i,j+1,k}^{n+1}}{h_y^2} + \frac{u_{i,j,k-1}^{n+1} - 2u_{i,j,k}^{n+1} + u_{i,j,k+1}^{n+1}}{h_z^2} \right\}$$

$$= u_{i,j,k}^n + 0.5\lambda dt \left\{ \frac{u_{i-1,j,k}^n - 2u_{i,j,k}^n + u_{i+1,j,k}^n}{h_x^2} + \frac{u_{i,j-1,k}^n - 2u_{i,j,k}^n + u_{i,j+1,k}^n}{h_y^2} + \frac{u_{i,j,k-1}^n - 2u_{i,j,k}^n + u_{i,j,k+1}^n}{h_z^2} \right\}$$

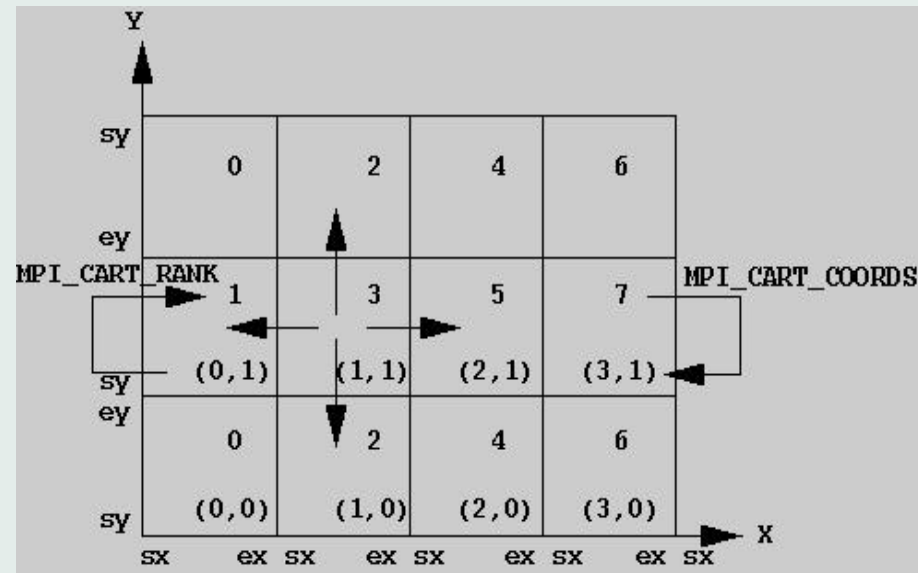
$$+ 0.5dt \left\{ f_{i,j,k}^{n+1} + f_{i,j,k}^n \right\}$$

11.2. Algorithme séquentiel

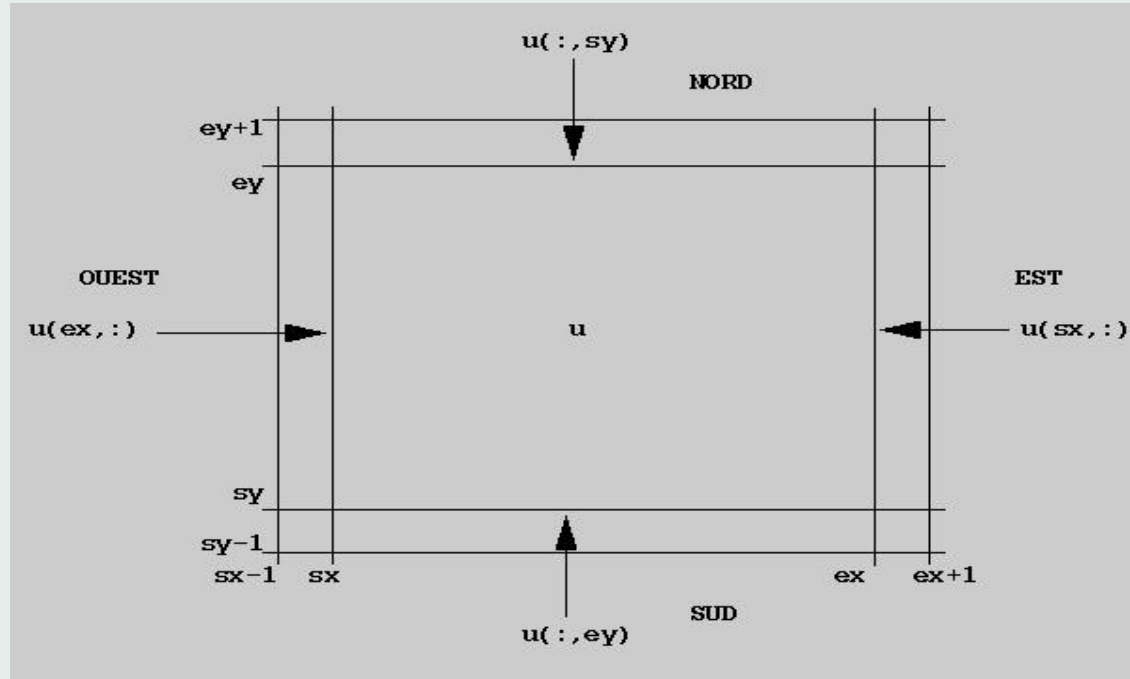
- Initialisations :
 - initialisations des vecteurs
- Boucle en temps
 1. Construction du second membre
 2. Résolution du système linéaire : **gradient conjugué**
 - Produit matrice-vecteur (pmv)
 - Produits scalaires (prodscal)
 - Combinaisons linéaires de vecteurs (saxpy)
 3. Avancement en temps
- Finalisations

11.3. Décomposition de domaine

- Partage du domaine de calcul en plusieurs morceaux, appelés sous-domaines
⇒ la somme directe forme le domaine initial
- Parallélisation explicite (travail à la charge du programmeur)
- Pour maillages structurés ou non structurés, avec des nœuds ou des cellules de calcul
- Grille de processus sur la grille des sous-domaines avec une topologie MPI
- Peut se faire quel que soit le nombre de processus
- Gestion du stockage et des synchronisations pour éviter les écrasements intempestifs.



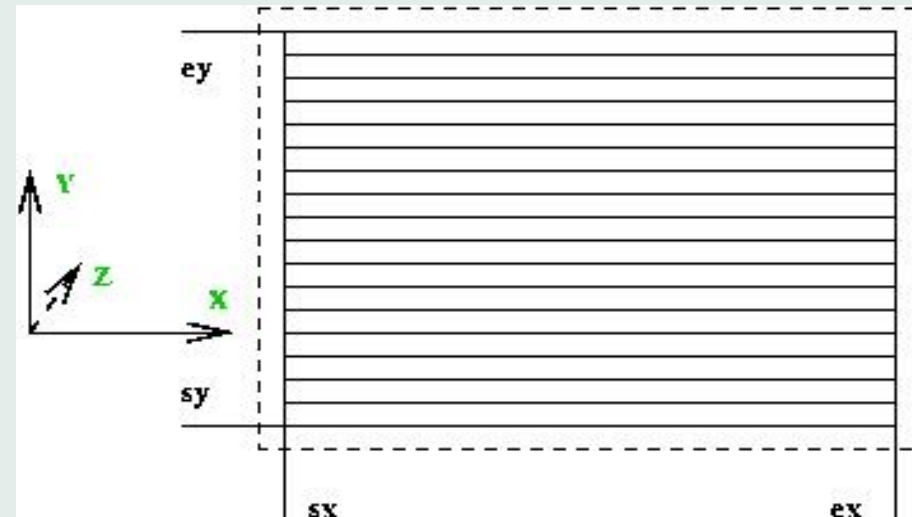
- Nœuds ou cellules fantômes nécessaires pour calculer les valeurs aux interfaces entre sous-domaines (conditions aux limites pour les bords extérieurs)
- Mise à jour de ces cellules à chaque pas de temps avec les valeurs calculées par le processus qui traite le sous-domaine propriétaire de ces cellules.
- Calcul automatique des dimensions locales, des espaces pour les nœuds fantômes, ...
indices locaux de calcul : de sx à ex ;
indices locaux de stockage : de $(sx-1)$ à $(ex+1)$ (**NB : longueur = $ex-sx+3$**)



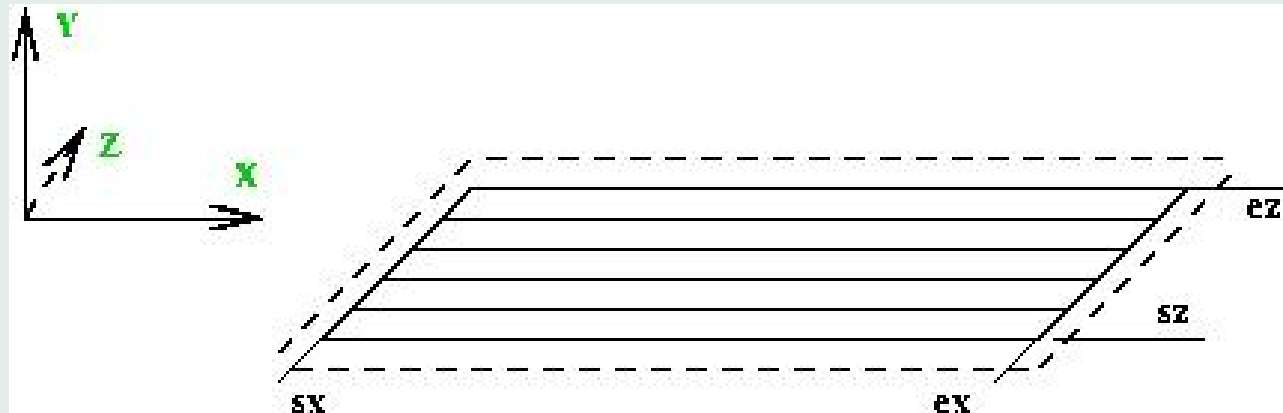
11.4. Types de données des messages

- Les sous-domaines sont des cubes
 ⇒ les zones d'interface des surfaces sont des sections des plans xOy , xOz et yOz
- Pour le plan xOy , il s'agit de $(ey-sy+1)$ vecteurs régulièrement espacés
 ⇒ longueur d'un vecteur : $ex-sx+1$,
 ⇒ pas entre deux débuts consécutifs : $ex-sx+3$
 ⇒ on crée le type `ip_xy` avec la fonction `MPI_TYPE_VECTOR` :

```
CALL MPI_TYPE_VECTOR(
ey-sy+1,      nombre de blocs
ex-sx+1,      longueur d'un bloc
(ex-sx+3),    pas entre le début de
ITYPE_REAL,   deux blocs consecutifs
ip_xy, ierr )
CALL MPI_TYPE_COMMIT(ip_xy, ierr)
```

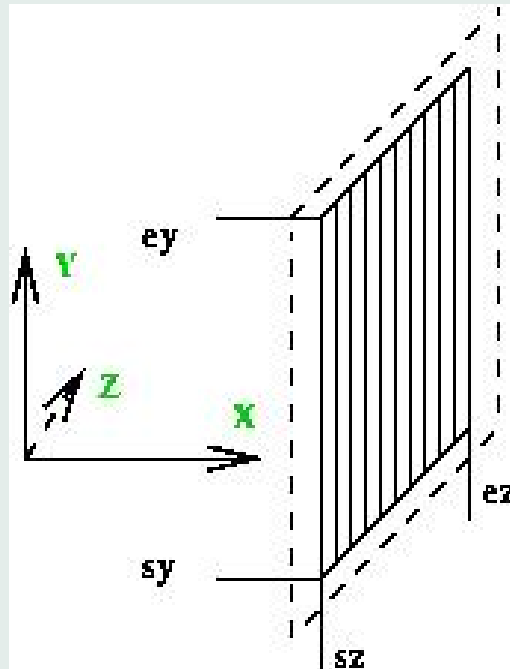


- Pour le plan \mathbf{xOz} , il s'agit de $(\mathbf{ez}-\mathbf{sz}+1)$ vecteurs régulièrement espacés
 - ⇒ longueur d'un vecteur : $\mathbf{ex}-\mathbf{sx}+1$,
 - ⇒ pas entre deux débuts consécutifs : $(\mathbf{ex}-\mathbf{sx}+3)*(\mathbf{ey}-\mathbf{sy}+3)$
 - ⇒ on crée le type **ip_xz** avec la fonction `MPI_TYPE_VECTOR`



```
CALL MPI_TYPE_VECTOR(
ez-sz+1,           nombre de blocs
ex-sx+1,           longueur d'un bloc
(ex-sx+3)*(ey-sy+3), pas entre le début de
ITYPE_REAL,       deux blocs consécutifs
ip_xz, ierr )
CALL MPI_TYPE_COMMIT(ip_xz, ierr)
```


- Pour le plan yOz , il s'agit d'éléments isolés irrégulièrement espacés
 - ⇒ en fait deux pas interviennent :
 - ⇒ Un pas constant dans la direction y et un pas constant dans la direction z
 - ⇒ On crée deux types dérivés : le second basé sur le premier en mesurant les distances entre les éléments en octets à l'aide de la fonction `MPI_TYPE_HVECTOR`



- extraction des "x" dans les "y" : type **type_y**

```
CALL MPI_TYPE_SIZE ( ITYPE_REAL, taille_real, ierr )
CALL MPI_TYPE_HVECTOR (
ey-sy+1,           nombre de blocs
1,                longueur d'un bloc
(ex-sx+3)*taille_real, pas entre le début de
ITYPE_REAL,       deux blocs consécutifs
type_y, ierr )
CALL MPI_TYPE_COMMIT ( type_y, ierr )
```

- extraction des "y" dans les "z" : type **ip_yz**

```
CALL MPI_TYPE_HVECTOR (
ez-sz+1,           nombre de blocs
1,                longueur d'un bloc
(ey-sy+3)*(ex-sx+3)*taille_real, pas entre le début de
type_y,           deux blocs consécutifs
ip_yz, ierr )
CALL MPI_TYPE_COMMIT ( ip_yz, ierr )
```

11.5. Communications

- synchronisation des processus :
 - ⇒ phase de calcul
 - ⇒ phase de communication
- échange entre les processus 2 à 2 : `MPI_SENDRECV`
 - ⇒ `MPI_SEND` et `MPI_RECV` en une seule communication

```
CALL MPI_SENDRECV(
a(sx,ey ,sz), 1, ip_xz, voisin(N), tag1,
a(sx,sy-1,sz), 1, ip_xz, voisin(S), tag1,
comm3d, status, ierr )
```

envoi au voisin N
réception du voisin S
type prédéfini ip_xz, plan xOz

```
CALL MPI_SENDRECV(
a(sx,sy ,sz), 1, ip_xz, voisin(S), tag2,
a(sx,ey+1,sz), 1, ip_xz, voisin(N), tag2,
comm3d, status, ierr )
```

envoi au voisin S
réception du voisin N
type prédéfini ip_xz, plan xOz

- de même pour les deux autres paires de faces

11.6. Algorithme MPI

- Initialisations :
 - construction de la topologie
 - construction des types MPI
 - initialisations des vecteurs locaux
 - diffusion des constantes / paramètres → communications
- Boucle en temps
 1. remplissage des cellules fantômes → communications
 2. construction du second membre
 3. résolution du système linéaire :
 - produit matrice-vecteur
 - produits scalaires
 - combinaisons linéaires de vecteurs (saxpy)
 - ⇒ communications / synchronisations
 4. avancement en temps
- Finalisations :
 - destruction des types MPI
 - destruction de la topologie

11.7. Partage du travail

- Distribution des itérations entre les processus
- parallélisation implicite (synchronisations, distribution des itérations gérées par la bibliothèque parallèle)
- le travail de parallélisation est transmis au compilateur au travers de directives insérées dans le code source.
- après analyse des dépendances, les boucles sont parallélisées ou non :
 - la boucle en temps : dépendance, **elle n'est pas parallélisable**
 - la boucle du gradient conjugué : dépendance, **elle n'est pas parallélisable**
 - les produits matrice-vecteur : pas de dépendance, ils sont parallélisables
 - les produits scalaires : pas de dépendance, ils sont parallélisables
 - les combinaisons linéaires de vecteurs : pas de dépendance, elles sont parallélisables
- création d'une région parallèle qui englobe la boucle en temps :
 - ⇒ toutes les threads l'effectuent
 - ⇒ toutes les threads effectuent la boucle du gradient conjugué
 - ⇒ toutes les threads se partagent les autres boucles

11.8. Algorithmes OpenMP

- Boucle en temps

```
!$OMP PARALLEL DEFAULT( NONE )
!$OMP SHARED(u,b,u_exact,f1,f2,p,q,r,rnorm2,rnormo)
!$OMP SHARED(tps,dt,rnorm_k,itgc,nbitgc,rerr2,erroo,prec)
!$OMP SHARED(nbproc,nbiter,machep,ifreq,it_max,rnudt2,rlambda)
!$OMP PRIVATE(itg)
DO itg = 1, nbiter
    CALL scdmb( b, f1, f2, u, tps, rnudt2 )
!$OMP MASTER
    nbitgc = nbitgc + itgc
    tps = tps + dt
    itgc = 0
    rnorm_k = zero
!$OMP END MASTER
    CALL gradconj( u, b, p, q, r, it_max, prec,
                  itgc, rnorm_k, rnudt2 )
END DO
!$OMP END PARALLEL
```

- Boucle de convergence du gradient conjugué

```

CALL pmv( r, u, - rnudt2 )
CALL saxpy( r, -one, b, one )
CALL prodscal( rnorm_k, r, r ) ← argument en sortie, donc partagé
it = 0 ← variable locale, donc privée
convergence = ( SQRT(rnorm_k) < prec )

DO WHILE ( (.NOT. convergence) .AND. (it < it_max) )
    it = it + 1
    CALL pmv( q, p, - rnudt2 )
    CALL prodscal ( alpha_k, q, p )
!$OMP MASTER
    itgc = it ← argument en sortie, donc partagé
    alpha_k = rnorm_k / alpha_k ← variables locales
    beta_k = rnorm_k initialisées, donc partagées
    rnorm_k = 0.0_rp ← argument en sortie, donc partagé
!$OMP END MASTER
!$OMP BARRIER

```

- Boucle de convergence du gradient conjugué (suite)

```
CALL saxpy( u, one, p, alpha_k )  
CALL saxpy( r, one, q, - alpha_k )  
CALL prodscal( rnorm_k, r, r )
```

```
!$OMP MASTER
```

```
alpha_k = 0.0_rp           ← variables locales initialisées  
beta_k = rnorm_k / beta_k ← donc partagées
```

```
!$OMP END MASTER
```

```
!$OMP BARRIER
```

```
CALL saxpy( p, beta_k, r, one )  
convergence = ( SQRT(rnorm_k) < prec )  
              ↑ variable locale, donc privée
```

```
END DO
```


- Boucles parallélisées : pmv

```

c1 = scal / hx**2           ← variable locale, donc privée
c2 = scal / hy**2           ← variable locale, donc privée
c3 = scal / hz**2           ← variable locale, donc privée
c4 = one - 2.0_rp * ( c1 + c2 + c3 )
!$OMP DO
DO k = sz, ez
  DO j = sy, ey
    DO i = sx, ex
      a(i,j,k) = c4 * b(i,j,k)
                + c1 * ( b(i+1,j ,k  ) + b(i-1,j ,k  ) )
                + c2 * ( b(i ,j+1,k  ) + b(i ,j-1,k  ) )
                + c3 * ( b(i ,j ,k+1) + b(i ,j ,k-1) )

    END DO
  END DO
END DO
!$OMP END DO

```

- Boucles parallélisées : prodscal et saxpy

```
!$OMP DO REDUCTION( +:rnorm )
DO k = sz, ez
  DO j = sy, ey
    DO i = sx, ex
      rnorm = rnorm + a(i,j,k) * b(i,j,k)
    END DO      ↑ argument en sortie, donc partagé,
  END DO      donc réduction
END DO
!$OMP END DO
```

```
!$OMP DO
DO k = sz, ez
  DO j = sy, ey
    DO i = sx, ex
      a(i,j,k) = scala * a(i,j,k) + scalb * b(i,j,k)
    END DO
  END DO
END DO
!$OMP END DO
```