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 (déboguage & analyse)
6. Optimisation scalaire
7. Calcul parallèle
8. Echange de messages avec MPI
9. Partage du travail avec OpenMP
10. Exemple récapitulatif

CRIHAN

1. Descriptions matérielle et logicielle

1.1. Description matérielle

Le cluster est composé de :

- 2 nœuds SMP p690 Turbo “Regatta” dotés chacun de :
  - 32 processeurs Power 4 (1.3 GHz, 5.2 GFlops),
  - 32 Go de mémoire vive,
  - 512 Go de disque interne.

  ⇒ **joe** (nœud de connection) et **jack** ;
- 1 switch Colony d'interconnection rapide entre les nœuds (512 Mo/s bi-directionnel) ;
- 1 baie de disques FastT500 (1 To de disque utile), reliée aux nœuds par des liens fibre optique (100Mo/s) ;
- 1 robot d'archivage ADIC-Grau Scalar 1000 (10 To de bandes magnétiques), dédié à la migration des données ;
- 1 console de supervision du cluster.

CRIHAN

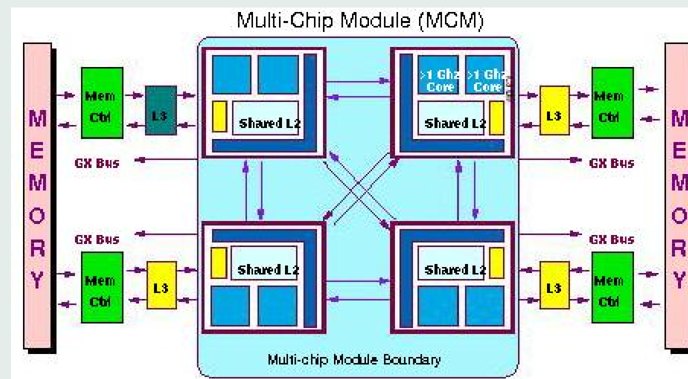
1.2. La puce Power 4

Les processeurs sont regroupés par deux au sein de la puce Power4 :
$\Rightarrow$ ils se partagent le cache de niveau 2 (L2), soit 1400 ko par puce.

Chaque processeur dispose de son propre cache de niveau 1 (L1),
soit 32 ko de données et 32 ko pour les instructions, par processeur.

1.3. Le Multi Chip Module (MCM)

Le bloc de construction de base pour un nœud p690 est le MCM (Multi Chip Module).
Composé de quatre puces Power4, il constitue ainsi un ensemble SMP de huit processeurs.



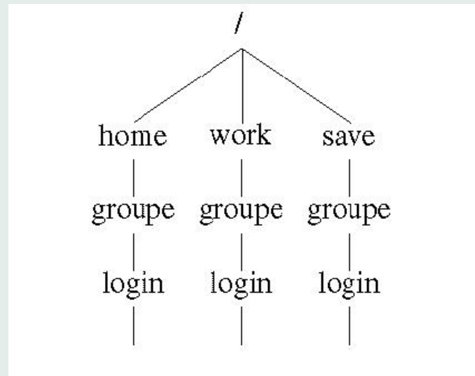
Les quatre MCM d'un même nœud sont reliés entre-eux point-à-point.
Ils disposent de 128 Mo de cache de niveau 3 (L3).

CRIHAN

1.4. 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`, (qui sera sauvegardé par la suite),
- un espace de travail `/work/projet/login`,
- un espace de migration `/save/projet/login`.



Les performances des Entrées / Sorties sont nettement moins bonnes lorsqu'elles ont lieu sur la baie FastT500.

⇒ Les exécutions des programmes en mode batch se font dans des espaces temporaires situés dans les disques internes des nœuds : chaque nœud dispose de 400 Go pour cela.

CRIHAN

1.5. Description logicielle

Système d'exploitation	AIX 5.1 , UNIX d'IBM
Compilateur Fortran	XLF, v. 8.1.1.3
Compilateur C	XLC, v. 6.0.0.5
Bibliothèques scientifiques	ESSL (inclus BLAS, FFT), v 3.3.0.6 ; LAPack, v 3.0 ; P-ESSL, v 2.3.0.2
Calcul parallèle	MPI, POE v 3.2.0.17 ; OpenMP ; PVM, v 3.4.3 (domaine public)
Gestionnaire de batch	LoadLeveler, v. 3.1.0.20
Logiciel de migration	TSM (non encore opérationnel)
Outils d'analyse	prof, gprof, xprofiler, PE Benchmarker (pct, pvt, ute), jumpshot
Débogueurs symboliques	dbx, pdbx
Editeurs de texte	vi ; emacs, v 20.7.1
Graphisme	gnuplot, v 3.7.2 ; xmgrace, v 5.1.11
Bibliothèque	netCDF (32 et 64 bits), v 3.5.0

CRIHAN

1.6. Connection et modes d'exécution

Les ressources du cluster sont séparées en deux parties :

(i) une partie du nœud **joe** pour la connection et l'interactif (de 4 à 8 processeurs, de 4 à 8 Go de mémoire) ;

(ii) une partie réservée au mode batch : le reste du nœud **joe** et le nœud **jack**.

Les connections, les éditions, les compilations, les exécutions en interactif prennent leurs ressources dans la partie **(i)**.
Des limites existent pour la mémoire, le temps consommé en interactif.

En général, les exécutions en mode batch prennent leurs ressources dans la partie **(ii)**, au sein d'un seul nœud :
⇒ Seuls les jobs MPI peuvent être à cheval sur les deux nœuds.

L'utilisateur exprime ses besoins en temps, processeurs, mémoire, dans un script.

Un espace temporaire est crée dans les disques internes (ou dans la baie) du nœud d'exécution et le système gère le transfert des fichiers.

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

2.1. Spécificités architecturales

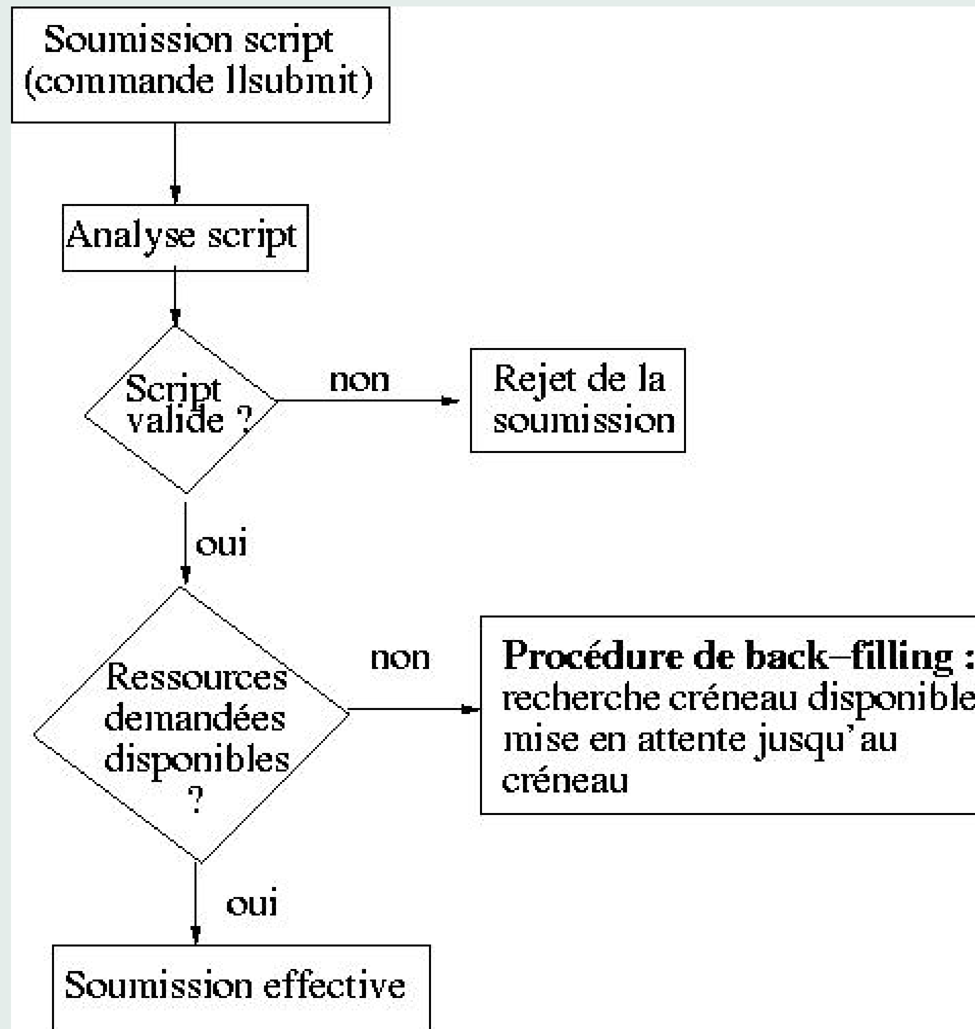
- Les comptes utilisateurs sont situés dans la baie externe FastT500
- Les débits vers la baie sont nettement moins performants que vers des disques internes
- Les exécutions ont lieu dans les disques internes du nœud (Attention avec MPI)

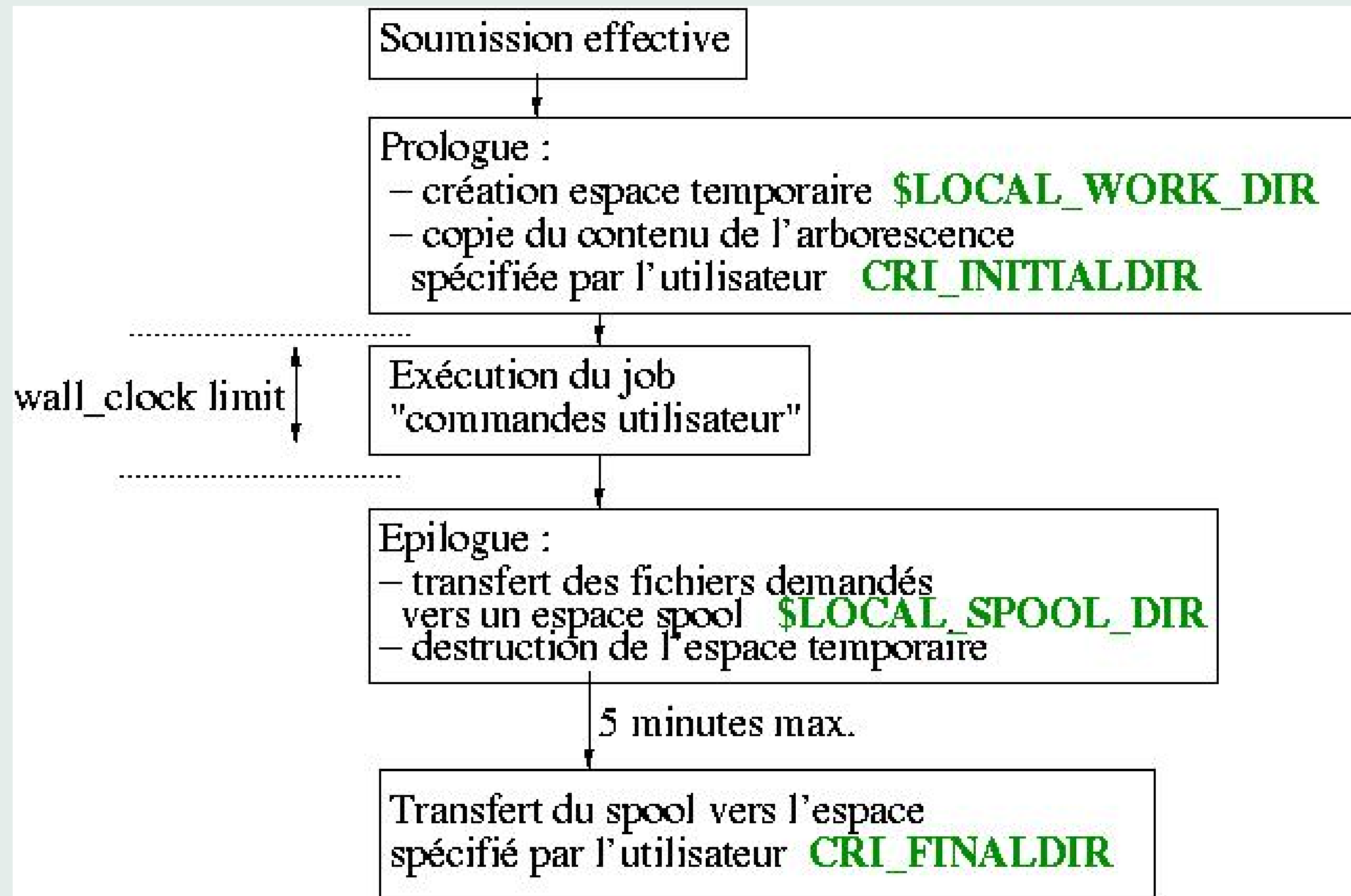
  ⇒ jobs MPI à cheval sur les nœuds par défaut

  ⇒ Le système gère les procédures de transfert de fichiers
- 10 Go d'espace disque alloué par processus

CRIHAN

2.2. Principe des soumissions





CRIHAN

$LOCAL_WORK_DIR représente le répertoire temporaire crée par le système dans les disques locaux du nœud et alloué au job pour son exécution ;
→ /local/run/ll-joe.xxxx.0 (ou /dlocal/run/ll-joe.xxxx.0)

CRI_INITIALDIR représente le répertoire spécifié par l'utilisateur dans son espace permanent sur la baie FastT500 dont le contenu est dupliqué dans les disques internes du nœud ;
→ /work/projet/login/REPERTOIRE_ENTREE

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

$LOCAL_SPOOL_DIR représente le répertoire temporaire crée par le système dans les disques locaux du nœud qui servent de transit avant le rapatriement vers l'espace permanent de l'utilisateur situé dans la baie FastT500.
→ /local/spool/ll-joe.xxxx.0 (ou /dlocal/spool/ll-joe.xxxx.0)

2.3. Mots-clés

Mot-clé	Séquentiel	MPI	OpenMP	Gaussian 98
`cri_job_type`	`serial`	`mpi`	`openmp`	`g98`
`cri_total_tasks`	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** ⇒ Attention à l'option `-bmaxdata`
- Liste non exhaustive ... (il existe aussi des scripts pour Gamess, Jaguar, ...)
- à compléter selon les besoins des utilisateurs
- **N.B.** : par défaut les jobs MPI sont à cheval sur les nœuds ⇒ `blocking`

2.4. Commandes

- soumission : `llsubmit` script_job → *ref_job* (ll-joe.xxxx.0)
- suivi : `llq`
- destruction : `llcancel` *ref_job*

CRIHAN

2.5. 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 = 256mb

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

CRIHAN

```
# Répertoire du compte utilisateur dont le contenu est copié
# @ cri_finaldir = /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
./a.out > ./OUT
# Déplacement des fichiers à récupérer
mv ./OUT $LOCAL_SPOOL_DIR
```

2.6. 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 sur 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 = 256mb

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

CRIHAN

```
# Répertoire du compte utilisateur dont le contenu est copié
# @ 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
./a.out > ./OUT
# Déplacement des fichiers à récupérer
mv ./OUT $LOCAL_SPOOL_DIR
```

2.7. 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 = 1024mb

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

CRIHAN

```
# Répertoire du compte utilisateur dont le contenu est copié
# @ 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
./a.out > ./OUT
# Déplacement des fichiers à récupérer
mv ./OUT $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	`xlf_r`	`xlf90_r`	`xlf95_r`
Code parallèle MPI	`mpxlf_r`	`mpxlf90_r`	`mpxlf95_r`
Code parallèle OpenMP, PVM, P-threads, ...	`xlf_r`	`xlf90_r`	`xlf95_r`

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

CRIHAN

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 des noms d'objets venant d'autres systèmes où ils sont absents (exemple : flush)

3.4. Options de compilation : déboguage

- **-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 commons, 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]

CRIHAN

3.5. Options de compilation : profilage / optimisation

- **-p[g]** : prépare le programme au profilage par `prof` ou `gprof`
- **-qreport=[hotlist|smplist]** fichiers listing des transformations du code
- **-O2**, **-O3** : niveau d'optimisation
- **-qstrict** : respect de la sémantique des programmes (si nécessaire)
- **-qhot** : optimisation plus aggressives (**-O3** nécessaire)
- **-qlargepage** : optimise pour des grandes pages (16 Mo au lieu de 4 ko)
- **-qipa** : analyse interprocédurale
- **-Q<x>** : inlining (**-Q**) ou non (**-Q!**), sélectif (**-Q**+*name1[:name2]* ou **-Q-***name1[:name2]*)
  → nécessite **-qipa** et au moins **-O2**
- **-qarch=auto** et **-qtune=auto** : optimisations spécifiques à l'architecture
- **-qsmp=omp** : parallélisation avec directives OpenMP

- Proposition d'options :
  -qarch=auto -qtune=auto -qhot -O3 -qstrict [-qsmp=omp]

CRIHAN

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*)
- **-brename:.name, .name_** : ajout d'un caractère souligné '_' à la fin d'un nom
- **-blpdata** : demande de larges pages (16 Mo) pour les applications consommatrices de mémoire

3.7. Analogies SGI MIPSpro / IBM XLF

Catégorie	Option SGI	Option IBM
Déboguage	-O0 -g -DEBUG:subscript_check=ON: trap_uninitialized=ON:div_check=3: verbose_runtime=ON	-qnooptimize -qcheck -qdbg -qextchk -qflttrap=ov:und:zero:inv:en -qfullpath -qinitauto=FF -qfloat=nans
Portage	-r8	-qdpc=e -qautodbl=dbl4
Optimisation	-r10000 -mips4 -O3 -OPT:IEEE_arithmetic=1:roundoff=0	-qarch=auto -qtune=auto -qhot -O3 -qstrict

Remarque : ⇒ **équivalence toute relative !**

CRIHAN

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://www.ibm.com/servers/eserver/pseries/library/sp_books/essl.html

CRIHAN

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 (et **prof**) : localisation de la consommation cpu

⇒ Compilation avec les options `-p[g] -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...]`

CRIHAN

Sortie de gprof : répartition du temps cpu

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
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	.prodscal [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.

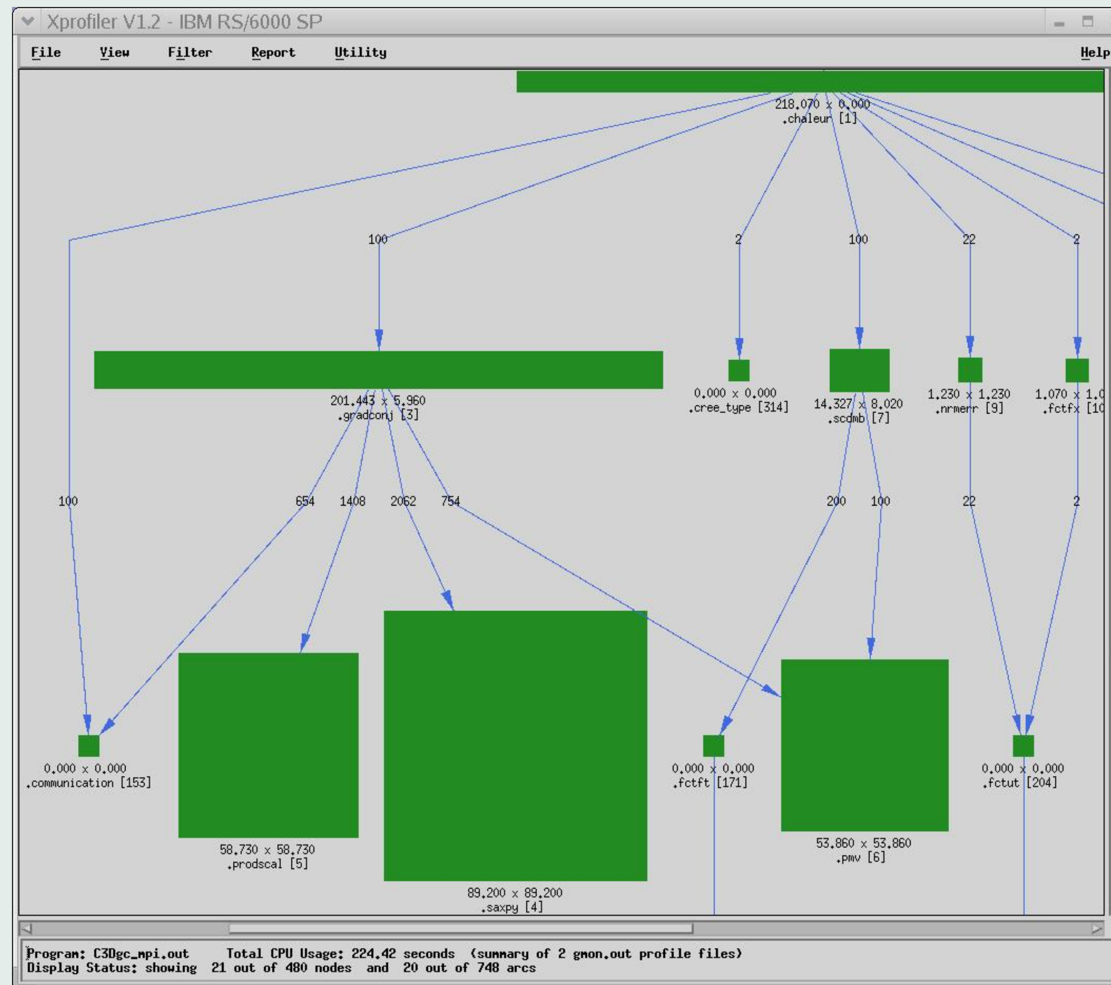
Sortie de gprof : hiérarchie des appels

index	%time	self	descendents	called/total called+self called/total	parents name index children
		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

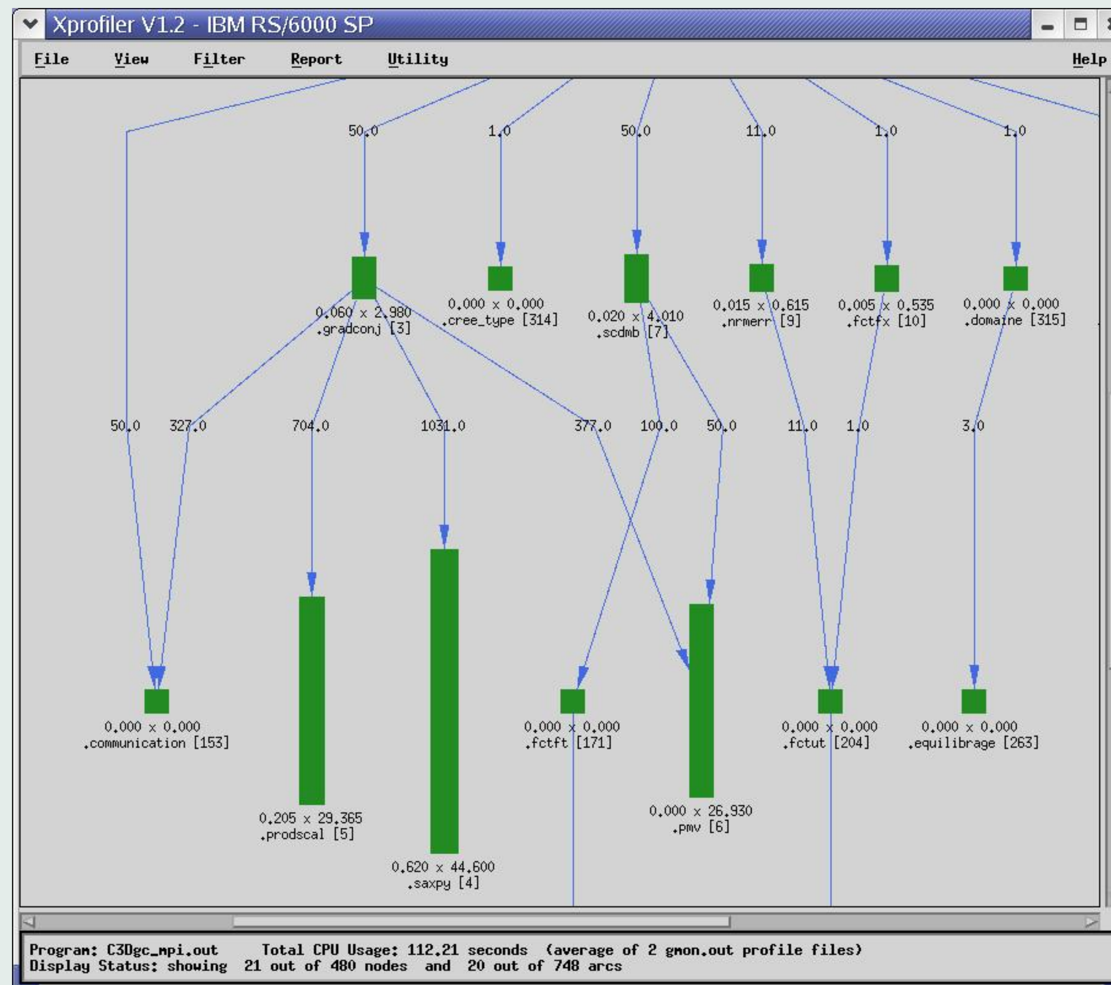
CRIHAN

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

CRIHAN

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

CRIHAN

PE Benchmarker

UTE Interval Files

utestats

utestats

Statistics Tables Generated By utestats Utility

node/cpu	0	1
0	2.823739	2.258315
1	0.873746	4.241253
2	0.956515	4.322891
3	0.853188	4.334650

AIX Trace Files

UTE Interval Files

uteconvert

uteconvert

uteconvert

uteconvert

uteconvert

uteconvert

utemerge

Merged UTE File

utestats

Trace Data Collection

Performance Collection Tool

UTE Interval Files

SLOG Files

slogmerge

slogmerge

Merged SLOG File

Jumpshot

(Note: Jumpshot is a public domain tool developed by Argonne National Laboratory, and is NOT part of the PE Benchmarker Toolset)

Hardware Performance Data Collection

netCDF Files

Profile Visualization Tool

6. Optimisation scalaire

6.1. Ecriture du code

- Réflexion au préalable sur papier
- Algorithmes performants (!)
- Déclaration explicite de toutes les variables, des dimensions des tableaux, des fonctions externes
  - Directive `IMPLICIT NONE` dans tous les fichiers source,
  - Type générique pour les variables (`REAL*8` ou `REAL(KIND=8)` 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

CRIHAN

6.2. 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]**

CRIHAN

6.3. 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

6.4. Validation du code

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

6.5. Analyse des performances

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

CRIHAN

6.6. Optimisation par re-écriture

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

6.7. 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 **X = 1** (**X = -1**) ; une multiplication avec **A = 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
```

CRIHAN

⇒ Calcul de puissances entières :

Version non optimisée

```
x = y ** 2.

ic = CMPLX (0., 1.   )

DO k = 1, n
  A(k) = ic **k * B(k)
END DO
```

Version optimisée

```
x = y **2

ic   = CMPLX (0.0_rp, 1.0_rp )
ic2 = ic
DO i = 1, n
  A(k) = ic2 * B(k)
  ic2  = ic2 * ic
END DO
```

⇒ 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

6.8. Arithmétique entière

- Ordre des opérations : $i * (i-1)/2$
  $i * \ \{(i-1)/2\} \neq \{i * (i-1)\}\ /2$
  $\Rightarrow$ Sans précaution, le niveau d'optimisation peut modifier le résultat
  $\Rightarrow$ Ne pas se limiter à un niveau d'optimisation inférieur mais éliminer ces instructions dangereuses

- Evaluation des multiplications / divisions par puissances de 2
  $\Rightarrow$ fonction **ISHFT** (entier, exposant signé)
  $\Rightarrow$ `ir * 8 = ISHFT( ir, 3 )` NB : $s^3 = 8$
  $\Rightarrow$ `(i * j * k) / 32 = ISHFT( i*j*k, -5 )` NB : $2^5 = 32$

CRIHAN

6.9. 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**)

6.10. Expressions

Ecrire des expressions identiques en spécifiant les variables dans le même ordre
⇒ Le compilateur Fortran sait reconnaitre des expressions identiques mais pas des permutations
⇒ **x = a + b + c + d** et **y = a + c + b + d** ne sont pas identiques ...
lorsque l'option **-qstrict** est présente

6.11. 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`, ...

CRIHAN

- 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
```

CRIHAN

- 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 !

- 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`

CRIHAN

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

++	+	-	--

`DO j = 1, M`
`  DO i = 1, N`
`    A(i,j)`
 	
`  END DO`
`END DO`
	`DO j = 1, M`
	`  DO i = 1, N`
	`    A(i+(j-1)*N)`
	
	`  END DO`
	`END DO`
		`DO j = 1, M`
		`  DO i = 1, N`
		`    k = k + 1`
		`    A(k)`
		`  END DO`
		`END DO`
			`DO j = 1, M`
			`  DO i = 1, N`
			`    A( index(i,j) )`
			
			`  END DO`
			`END DO`

- 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
```

... 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
```

⇒ 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 !

CRIHAN

- Déroulage des boucles (*unrolling*)
  boucles internes → augmenter la masse d'opérations indépendantes par itération
  boucles externes → augmenter le ratio FMA / load-store par itération :

```
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 ↦ 5 loads / 4 FMA

s0, s1, s2, s3 :
scalaires temporaires importants pour limiter les MàJ intempestives de `y(i)`, ..., `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
```

CRIHAN

- Re-employer les données présentes dans le cache (*cache blocking*)
  ⇒ travailler sur des blocs de données qui rentrent dans le cache de données

  ⇒ augmentation des performances si :

  1. nombre d'itérations $\ggg$ nombre de données utilisées
  2. boucles permutables,
  3. taille des blocs $<$ taille du cache disponible

Exemple simple : produit matrice-matrice $\mathbf{D} = \mathbf{A} \quad \mathbf{B}$

```
DO i = 1, n
  DO j = 1, n
    DO k = 1, n
      d(i,j) = d(i,j) + a(j,k) × b(k,i)
    END DO
  END DO
END DO
```

CRIHAN

Conclusions :

- le compilateur ne fait pas tout
- dans des cas simples, il peut s'en sortir
- dans des situations plus complexes, le travail du développeur est incontournable
- utiliser les outils de profilage pour :
  - localiser les zones consommatrices
  - déterminer l'optimisation la plus efficace
- utiliser les fichiers de listing générés par le compilateur
- toujours valider la nouvelle version du code !

CRIHAN

7. Calcul parallèle : Quelques définitions

7.1. 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, ...

7.2. 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).

7.3. 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.

7.4. Loi d'Amdhal

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

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

seq : portion séquentielle du code
par : portion parallèle du code
} $par + seq = 1$
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 $A(par, seq) < 1/seq = 10 \quad \forall N$

CRIHAN

8. 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

CRIHAN

8.1. Environnement

- Initialisation en début (MPI_INIT)
- Finalisation en fin de programmme (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

CRIHAN

8.2. Structures de données

- Les données transmises sont typées
- Types prédéfinis : MPI_INTEGER, MPI_REAL, ...
- Type homogène :
  - données contigües : 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

CRIHAN

8.3. 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

CRIHAN

8.4. 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 :

8.4.1. les synchronisations globales

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

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

8.4.2. les opérations de réduction sur des données réparties

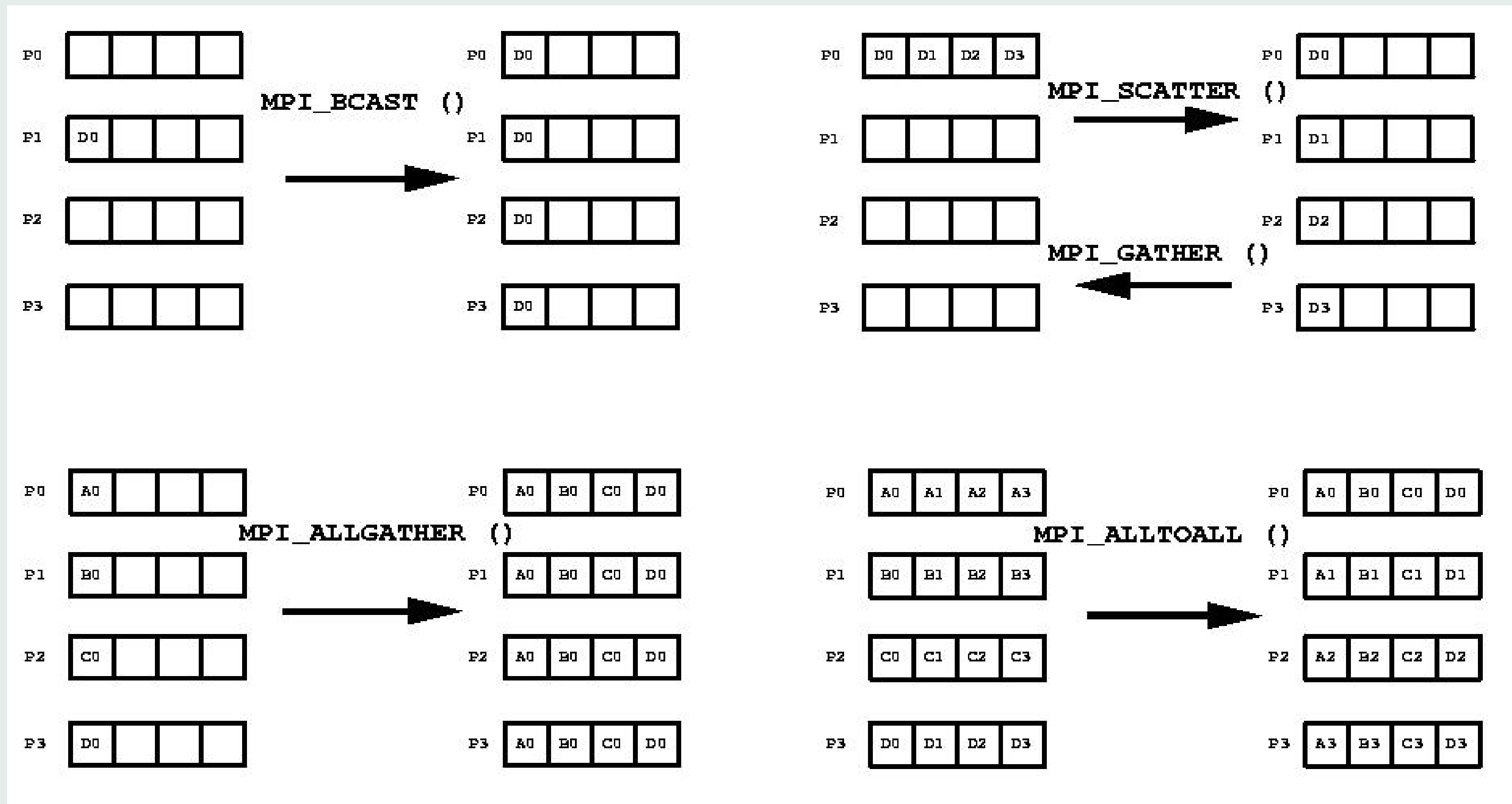
somme, produit, maximum ... effectué sur des données réparties (MPI_REDUCE) et le résultat peut être ensuite redistribué (MPI_ALLREDUCE).

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

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

CRIHAN

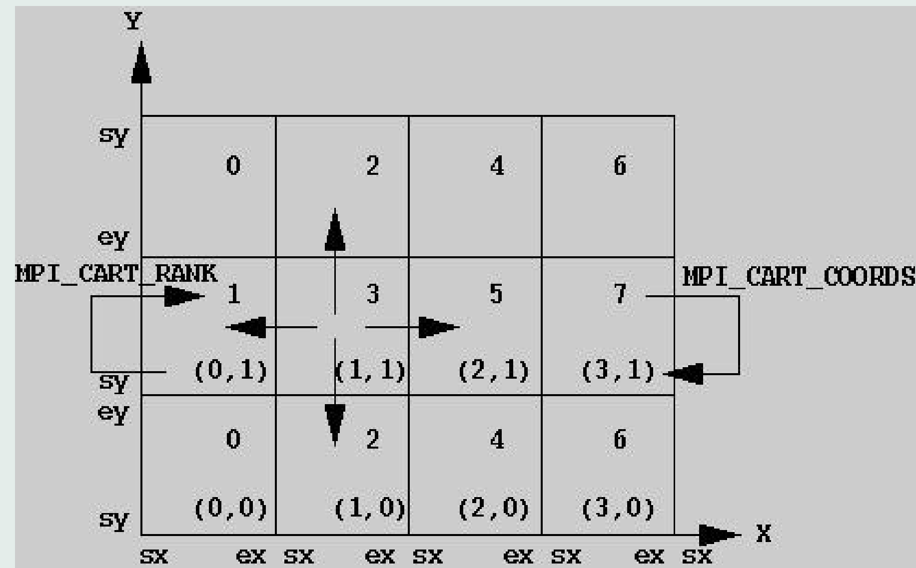
8.4.3. Diffusion / collecte



CRIHAN

8.5. 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`



8.6. 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

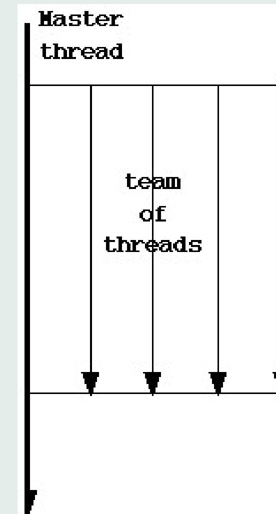
- Exemple récapitulatif : équation de la chaleur en 3D par décomposition de domaine
  http://www.crihan.fr/calcul/IBM_DOC_WEB/EDP_paralleles/c3d/chaleur.html

CRIHAN

9. 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



CRIHAN

9.1. 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

CRIHAN

9.2. 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, ...

CRIHAN

9.3. 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.

9.4. 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, ...

CRIHAN

9.5. 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]`

CRIHAN

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
```

CRIHAN

9.6. Sérialisation et synchronisations

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

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

CRIHAN

9.7. 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/IBM_DOC_WEB/EDP_paralleles/c3d/chaleur.html

CRIHAN

10. Exemple récapitulatif : équation de la Chaleur 3D

10.1. définition du problème

- Les équations ...

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

- 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
  $\Rightarrow$ maillage uniforme noté x_i, j, k
  $\Rightarrow$ 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.

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

CRIHAN

- Le laplacien est ainsi discrétisé :

$$\Delta_h u(i,j,k) = \frac{u_{i-1,j,k} - 2.u_{i,j,k} + u_{i+1,j,k}}{{h_x}^2} + \frac{u_{i,j-1,k} - 2.u_{i,j,k} + u_{i,j+1,k}}{{h_y}^2}$$

$$+ \frac{u_{i,j,k-1} - 2.u_{i,j,k} + u_{i,j,k+1}}{{h_z}^2}$$

(termes d'erreurs en ${h_x}^2, {h_y}^2, {h_z}^2$).

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

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

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

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

CRIHAN

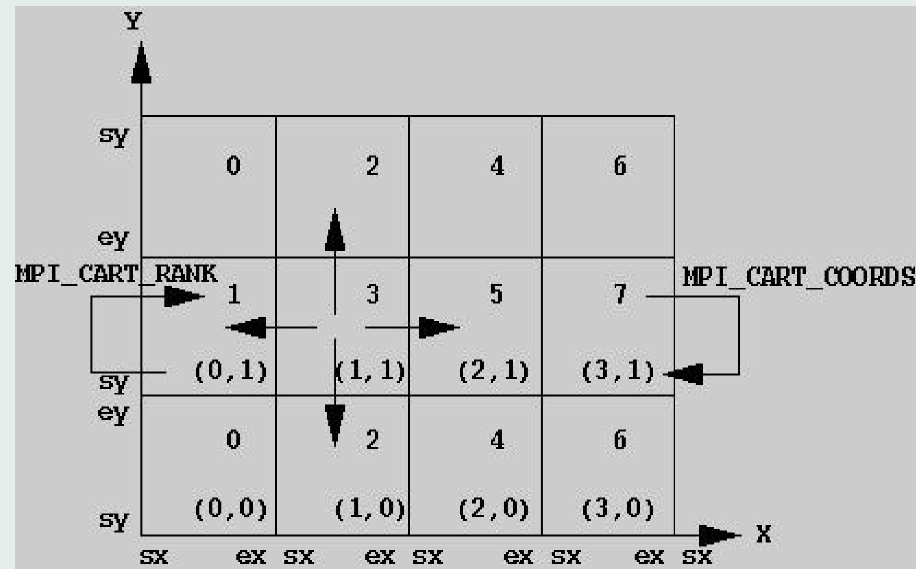
10.2. Algorithme séquentiel

- 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

CRIHAN

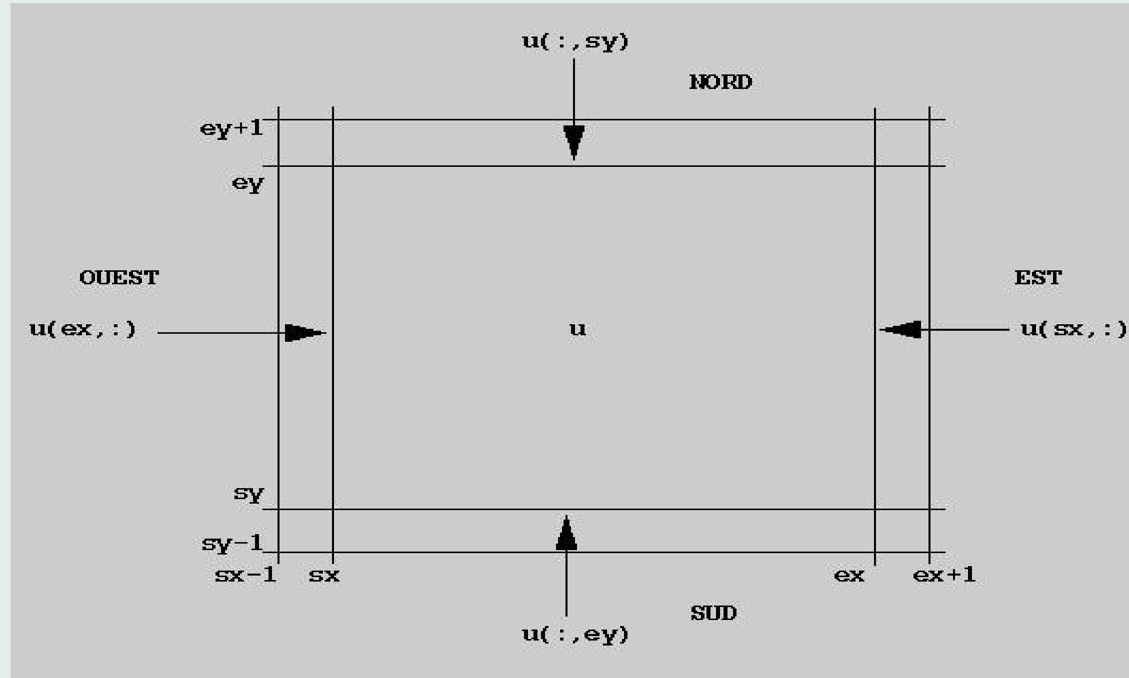
10.3. Décomposition de domaine

- Partage du domaine de calcul en plusieurs morceaux, appelés sous-domaines
  $\Rightarrow$ 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.



CRIHAN

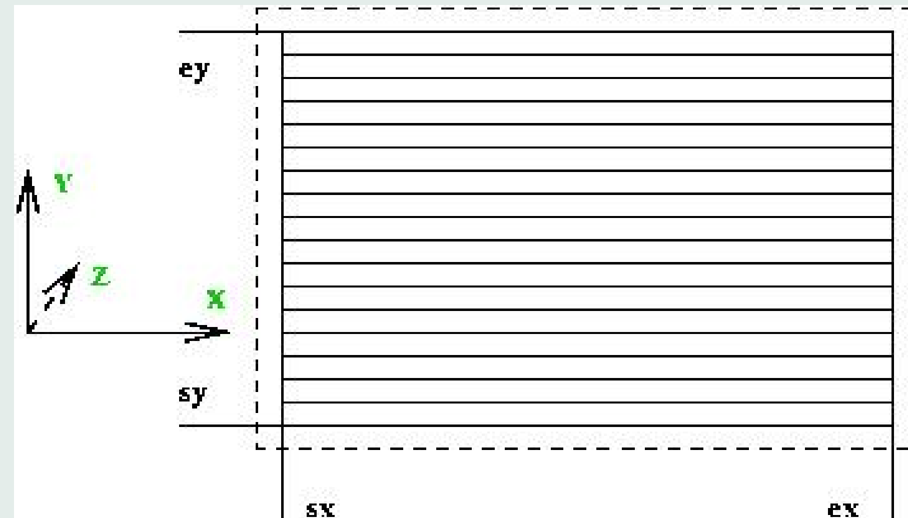
- Nœuds ou cellules fantômes nécessaires pour calculer les valeurs aux interfaces entre sous-domaines (conditions aux limites pour les bords extérieurs)
- Mise à jour de ces cellules à chaque pas de temps avec les valeurs calculées par le processus qui traite le sous-domaine propriétaire de ces cellules.
- 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)



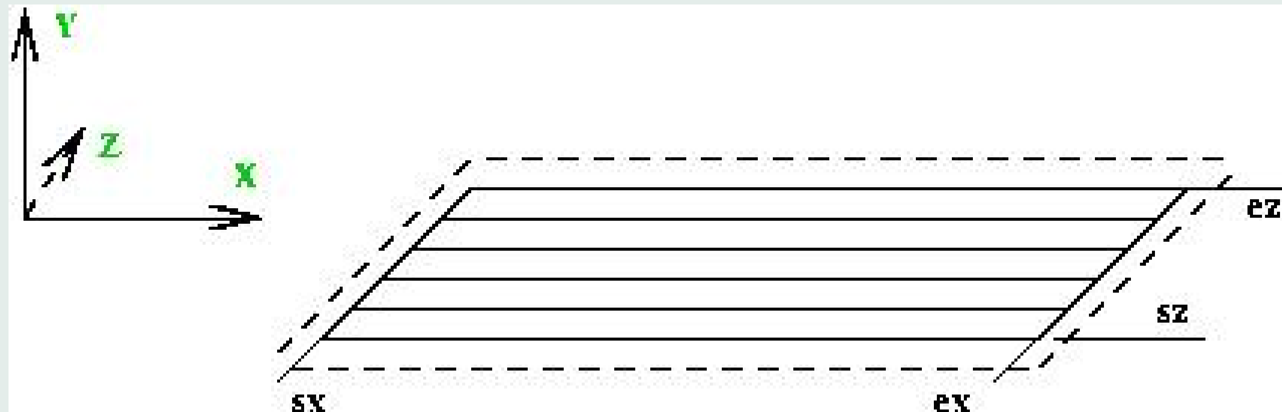
10.4. Types de données des messages

- 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)
```



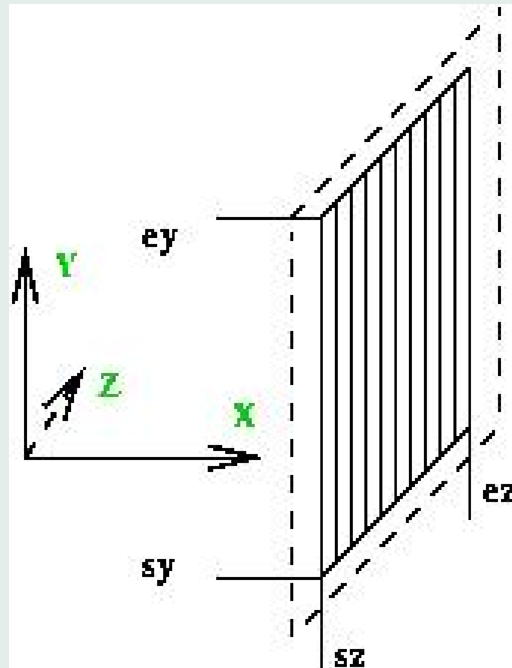
- Pour le plan **xOz**, il s'agit de (**ez-sz+1**) vecteurs régulièrement espacés
  ⇒ longueur d'un vecteur : **ex-sx+1**,
  ⇒ pas entre deux débuts consécutifs : **(ex-sx+3)*(ey-sy+3)**
  ⇒ on crée le type **ip_xz** avec la fonction `MPI_TYPE_VECTOR`



`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)`	

CRIHAN

- 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 )
```

CRIHAN

10.5. 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,   envoi au voisin N
a(sx,sy-1,sz), 1, ip_xz, voisin(S), tag1,   réception du voisin S
comm3d, status, ierr )                      type prédéfini ip_xz, plan xOz

CALL MPI_SENDRECV(
a(sx,sy  ,sz), 1, ip_xz, voisin(S), tag2,   envoi au voisin S
a(sx,ey+1,sz), 1, ip_xz, voisin(N), tag2,   réception du voisin N
comm3d, status, ierr )                      type prédéfini ip_xz, plan xOz
```

- de même pour les deux autres paires de faces

CRIHAN

10.6. 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

CRIHAN

10.7. 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

CRIHAN

10.8. Algorithmes OpenMP

- Boucle en temps

```
!$OMP PARALLEL DEFAULT( NONE )
!$OMP SHARED(u,b,u_exact,f1,f2,p,q,r,rnrm2,rnrmoo)
!$OMP SHARED(tps,dt,rnorm_k,itgc,nbitgc,rerr2,rerroo,prec)
!$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
```

CRIHAN

- 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
```

CRIHAN

- 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
```

CRIHAN

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

CRIHAN

Conclusions

- Le portage et le développement se font en suivant des règles :
  → respect des normes du langage,
  → utilisation d'outils d'analyse,
  → options de compilation adaptées.

- Optimisation scalaire : par re-écriture et options de compilation

- MPI : standard portable pour architecture à mémoire distribuée
  ⇒ parallélisme à gros grains
  ⇒ développement obligatoire de toute l'application

- OpenMP : standard portable pour architecture à mémoire partagée
  ⇒ parallélisme à gros grains et petits grains
  ⇒ développement incrémental basé sur la sémantique du code séquentiel

- Optimisations parallèles : choisir la technique selon le cas et la machine