Formation théorique : environnement de travail sur la grappe IBM

Guy MOEBS - CRIHAN
Février 2007

Plan

1. Description matérielle et logicielle
2. Soumission des travaux
3. Compilateurs
4. Environnement logiciel scientifique
5. Optimisation scalaire
6. Calcul parallèle

Description matérielle et logicielle

Description matérielle

Les éléments qui constituent la grappe IBM sont :

- les nœuds de calcul p575 (Power 5)
- le réseau d'interconnexion rapide
- les frontales de connexion
- les nœuds de calcul p690 (Power 4)
- les espaces de stockage

- 22 nœuds de calcul SMP p575 :
  - 8 processeurs Power 5 (1.9 GHz, 7.6 GFlops)
  - 16 Go de mémoire vive
- réseau d'interconnexion rapide *Federation*
  - 1 Go/s, bi-directionnel
  - 16 nœuds p575 interconnectés
- 2 frontales de connexion p510, *william* et *averell* :
  - 2 processeurs Power 5 (1.5 GHz)
  - 4 Go de mémoire vive

- 2 nœuds de calcul SMP p690 :
  - 32 processeurs Power4 (1.3 GHz, 5.2 GFlops)
  - 32 Go de mémoire vive
- 22 To de disque utile

Fonction	Localisation	Quota utilisateur	Caractéristique	Capacité
espace utilisateur	/home/projet/login (HOME_DIR)	5 Go	RAID 5 + sauvegarde	1.2 To
espace de travail	/work/projet/login (WORK_DIR)	25 Go	RAID 5	4.0 To
espace de migration	/save/projet/login (SAVE_DIR)	300 Go	RAID 5	7.0 To
espaces temporaires (LL)	/dlocal/run/LL-id /dlocal/spool/LL-id	1 To	RAID 0	8.0 To

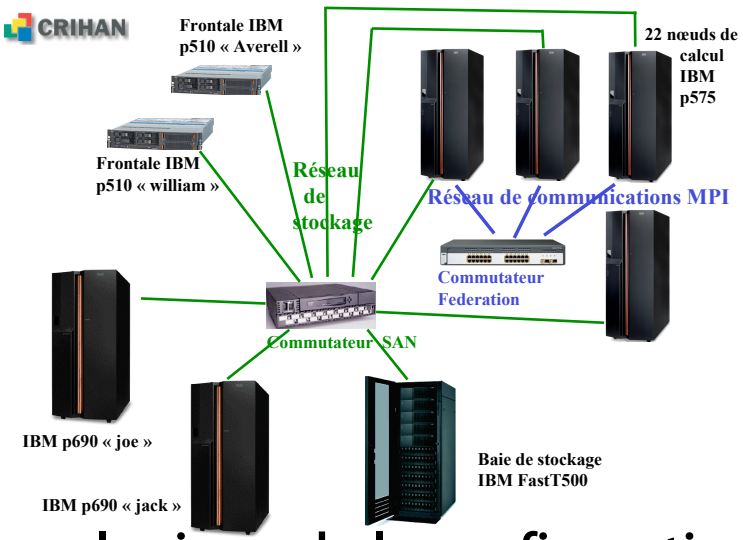
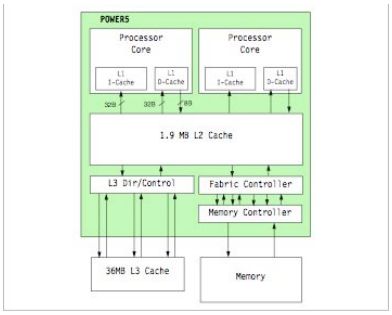


schéma logique de la configuration

La puce Power 5

- Elle est à double cœur mais un seul est activé
- Elle dispose d'une hiérarchie de cache L1, L2, L3
- Le contrôleur du L3 est désormais intégré au processeur
- Chaque p575 compte 8 puces



Caractéristiques de la hiérarchie mémoire

Latences	Registres	cache L1	cache L2	cache L3	mémoire
Power 4	1	2	12	123	351
Power 5	1	2	13	87	235

	Power 4	Power 5
cache L1	2 voies associatives, FIFO 32 ko (D) + 64 ko (I) ligne de 128 octets	4 voies associatives, LRU 32 ko (D) + 64 ko (I) ligne de 128 octets
cache L2	4 voies associatives 1440 ko (par double cœur) ligne de 128 octets	10 voies associatives 1920 ko (par double cœur) ligne de 128 octets
cache L3	8 voies associatives 32 Mo ligne de 256 octets	12 voies associatives 36 Mo ligne de 256 octets

Configuration mémoire

- Chaque nœud p575 dispose de 16 Go de mémoire physique

➡ environ 12 Go de mémoire disponible pour les applications

- Chaque nœud est configuré en 10 Go (LP) + 2 Go (SP)
- LP (Large Page) : page mémoire de 16 Mo

➡ performances accrues

- SP (Small Page) : page mémoire de 4 ko
- Faire attention à l'édition des liens
- Faire attention dans les scripts de soumission

Description logicielle

Système d'exploitation	AIX 5.3, UNIX IBM
Compilateur Fortran	XLF 10.1.0.2
Compilateur C, C++	XLC 8.0.0.6, vac C++ 8.0.0.11, gcc/g++ 4.0.0
Bibliothèques scientifiques	ESSL 4.2.0.4, P-ESSL 3.2.0.1, LAPack 3.0, ScaLAPack, BLACS, FFTW 3.1.2
Calcul parallèle	MPI, OpenMP
Gestionnaire de traitement par lots	LoadLeveler 3.3.2.7
Outils d'analyse	gprof, xprofiler, PE Benchmarker (pct, pvt)
Débogueurs symboliques	dbx, pdbx, gdb
Editeurs de texte	vi, emacs, nedit
Graphisme	gnuplot 4.0.0, xmgrace 5.1.11, pgplot, ncview
Bibliothèques diverses	netCDF (3.6.0), NCO (3.1.2), HDF (4.2.1, 5.1.6)

Logiciels 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. 3.11.1
  - Yale University : CNS v. 1.1
  - Iowa State University : Gamess (16/02/2002)
- On peut installer vos logiciels commerciaux sous licence ...

Connexion et modes d'exécution

- Les ressources de la grappe sont séparées en deux parties :

(i) les frontales : la connexion, 2 p510

(ii) les nœuds de calcul : dédiés au traitement par lots, 22 p575 et 2 p690

- Les connexions, éditions, compilations et mises au point prennent leurs ressources dans (i)

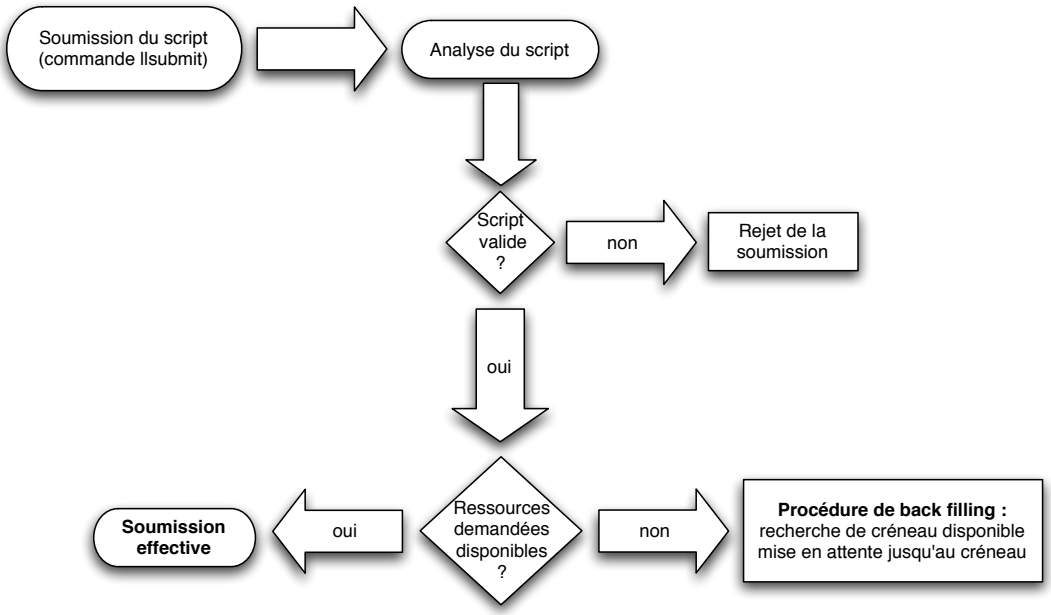
- Les travaux soumis à LoadLeveler prennent leurs ressources dans (ii)

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

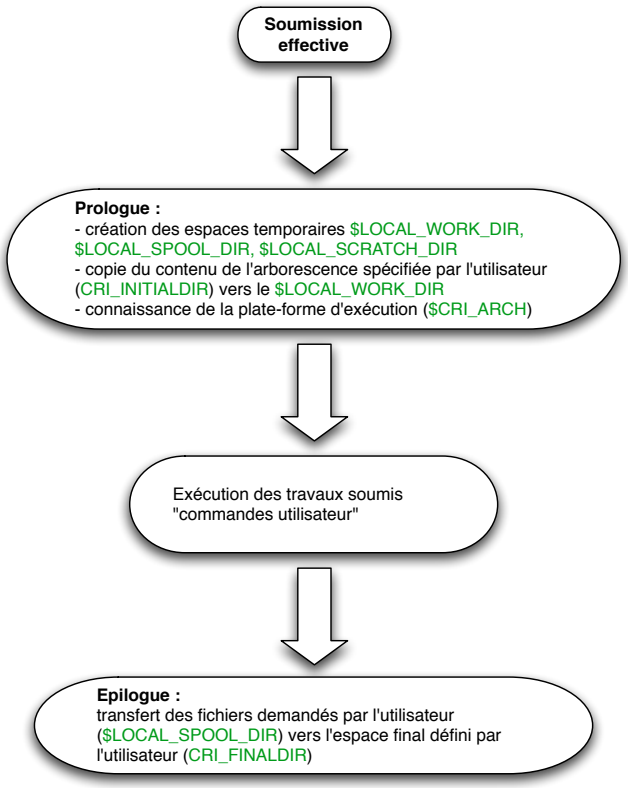
Soumission de travaux au travers de LoadLeveler

Mode de fonctionnement

- La soumission se fait au travers de scripts
- L'expression des besoins se fait au travers de directives
- Les directives ont la forme **# @ directive**
- L'expression des besoins pondère les soumissions et décide des priorités
- Les travaux parallèles MPI sont à cheval entre les nœuds par défaut
- Il faut fixer les quantités de mémoire *data* (LP) et *stack* (SP)



Principe de soumission



Commentaires .....

- Répertoires permanents de l'utilisateur :
  - le CRI_INITIALDIR représente le répertoire spécifié par l'utilisateur dans son espace permanent dont le contenu est dupliqué dans le LOCAL_WORK_DIR

  ➡ exemple : /work/projet/login/REPERTOIRE_ENTREE

  - le CRI_FINALDIR représente le répertoire spécifié par l'utilisateur dans son espace permanent dans lequel les fichiers spécifiés par l'utilisateur sont rapatriés par le système :

  ➡ exemple : /work/projet/login/REPERTOIRE_SORTIE

  - Ils doivent respecter les règles suivantes :
    - être des chemins absolus dans le WORK_DIR ou le HOME_DIR
    - ne contenir que des fichiers réguliers
    - le CRI_INITIALDIR ne doit pas être vide
    - le CRI_FINALDIR est crée s'il n'existe pas

Répertoires temporaires crées par LoadLeveler

- le LOCAL_WORK_DIR représente le répertoire d'exécution, visible par tous les nœuds de la grappe

➡ /dlocal/run/ll-william.xxxxx.0 ou /dlocal/run/ll-averell.yyyyy.0

- le LOCAL_SPOOL_DIR représente le répertoire de transit, visible par tous les nœuds de la grappe. L'utilisateur y déplace les fichiers du LOCAL_WORK_DIR qu'il souhaite rapatrier

➡/dlocal/spool/ll-william.xxxxx.0 ou /dlocal/spool/ll-averell.yyyyy.0

- le LOCAL_SCRATCH_DIR représente un répertoire de travail local au nœud. Son usage est restreint aux applications intra-nœud.

Sur quels nœuds sont exécutés mes travaux ?

- La grappe est une plate-forme hétérogène
- Les travaux non spécifiques à une plate-forme sont susceptibles d'être exécutés sur tous les nœuds
- LoadLeveler attribue les ressources selon ses disponibilités
- L'optimisation d'un code à une plate-forme améliore ses performances sur celle-ci mais qu'en est-il sur une autre ?

➡ un code optimisé Power 4 s'exécute sur Power 5
➡ un code optimisé Power 5 peut ne pas s'exécuter sur Power 4
➡ un code optimisé Power 4 est moins efficace qu'un code optimisé Power 5 sur Power 5 (dégradation variable, selon l'application)

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

Korn Shell	C Shell

if [$CRI_ARCH = "pwr5"]
then
  ./a.out_pwr5
else
  ./a.out_pwr4
fi
	if ($CRI_ARCH = "pwr5")
	then
	  ./a.out_pwr5
	else
	  ./a.out_pwr4
	endif

Directives et mots-clefs

- Une directive est formée d'un mot-clef et de sa valeur

Mot-clef	Séquentiel	MPI	OpenMP	Gaussian
cri_job_type	serial	mpi	openmp	gaussian
cri_total_tasks	sans objet	nombre de processus MPI	nombre de threads OpenMP	%NProc

- wall_clock_limit : temps maximal d'exécution du travail
- data_limit : mémoire maximale par processus (!)

  ➡mémoire prise dans les Large Pages
- stack_limit : mémoire maximale par processus (!)

  ➡ optionnel ; 256 Mo par défaut ; mémoire prise dans les Small Pages
- core_limit (0 Mo par défaut) : taille maximale par processus

Commandes

- Soumission : llsubmit script_job ➡ (réf. william-a1.uvwxy.0)

➡ llsubmit ll_mpi

llsubmit: Processed command file through Submit Filter: "/soft/loadl/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
```

- Destruction : llcancel william-a1.uvwxy.0

➡ llcancel william-a1.2045.0

llcancel: Cancel command has been sent to the central manager.

Script séquentiel (1/3)

```
#!/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 = 0:30:00

# Memoire maximale par processus (mb, gb, mw, gw, ..)
# @ data_limit  = 350mb
# Stack maximale par processus (mb)
# @ stack_limit = 100mb
```

Script séquentiel (2/3)

```
# Repertoire initial a envoyer
# @ cri_initialdir = /work/projet/mon_login/TEST_SEQ_IN

# Repertoire final pour les resultats
# @ cri_finaldir = /work/projet/mon_login/TEST_SEQ_OUT

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

# Obligatoire
# @ queue
```


Script séquentiel (3/3)

```
###
### Commandes utilisateur
###
# On se positionne dans le repertoire temporaire de calcul : $LOCAL_WORK_DIR
cd $LOCAL_WORK_DIR

# Execution du programme sequentiel
if (CRI_ARCH=pwr5) then
  ./a.out_pwr5 > $LOCAL_SPOOL_DIR/out_seq
else
  ./a.out_pwr4 > $LOCAL_SPOOL_DIR/out_seq
endif

# On deplace les fichiers dans le repertoire temporaire de rapatriement des donnees
mv ./resultat.dat $LOCAL_SPOOL_DIR
```

Script parallèle MPI (1/3)

```
#!/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
# @ cri_total_tasks = 6

# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 0:30:00
```


Script parallèle MPI (2/3)

```
# Memoire maximale par processus (mb, gb, mw, gw,..)
# @ data_limit  = 550mb
# Stack maximale par processus (mb)
# @ stack_limit = 100mb

# Repertoire initial a envoyer
# @ cri_initialdir = /work/mon_projet/mon_login/TEST_MPI_IN
# Repertoire final pour les resultats
# @ cri_finaldir = /work/mon_projet/mon_login/TEST_MPI_OUT

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

# Obligatoire
# @ queue
```

Script parallèle MPI (3/3)

```
###
### Commandes utilisateur
###
# On se positionne dans le repertoire temporaire de calcul : $LOCAL_WORK_DIR
cd $LOCAL_WORK_DIR

# Execution du programme parallele MPI
if (CRI_ARCH=pwr5) then
  ./a.out_pwr5 > $LOCAL_SPOOL_DIR/out_mpi
else
  ./a.out_pwr4 > $LOCAL_SPOOL_DIR/out_mpi
endif

# On deplace les fichiers dans le repertoire temporaire de rapatriement des donnees
mv ./resultat.dat $LOCAL_SPOOL_DIR
```


Script parallèle OpenMP (1/3)

```
#!/bin/csh
# Script de soumission Loadleveler, job parallele 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 de threads OpenMP
# @ cri_total_tasks = 6

# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 0:30:00
```

Script parallèle OpenMP (2/3)

```
# Memoire maximale POUR l'application (mb, gb, mw, gw, ..)
# @ data_limit  = 850mb
# Stack maximale POUR l'application (mb)
# @ stack_limit = 400mb

# Repertoire initial a envoyer
# @ cri_initialdir = /work/projet/mon_login/TEST_OMP_IN
# Repertoire final pour les resultats
# @ cri_finaldir = /work/projet/mon_login/TEST_OMP_OUT

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

# Obligatoire
# @ queue
```


Script parallèle OpenMP (3/3)

```
###
### Commandes utilisateur
###
# On se positionne dans le repertoire temporaire de calcul : $LOCAL_WORK_DIR
cd $LOCAL_WORK_DIR

# Execution du programme parallele openmp
if (CRI_ARCH=pwr5) then
  ./a.out_pwr5 > $LOCAL_SPOOL_DIR/out_omp
else
  ./a.out_pwr4 > $LOCAL_SPOOL_DIR/out_omp
endif

# On deplace les fichiers dans le repertoire temporaire de rapatriement des donnees
mv ./resultat.dat $LOCAL_SPOOL_DIR
```

Script parallèle Gaussian (1/3)

```
#!/bin/csh
# Script de soumission Loadleveler, job parallele Gaussian G03 version c02
# 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 de threads Gaussian
# @ cri_total_tasks = 4

# temps de restitution (heures[:minutes[:secondes]])
# @ wall_clock_limit = 0:30:00
```


Script parallèle Gaussian (2/3)

```
# Memoire maximale POUR l'application (mb, gb, mw, gw, ..)
# @ data_limit  = 1600mb
# Stack maximale POUR l'application (mb)
# @ stack_limit = 400mb

# Repertoire initial a envoyer
# @ cri_initialdir = /work/projet/mon_login/TEST_G03_IN
# Repertoire final pour les resultats
# @ cri_finaldir = /work/projet/mon_login/TEST_G03_OUT

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

# Obligatoire
# @ queue
```

Script parallèle Gaussian (3/3)

```
###
### Commandes utilisateur
###
# On se positionne dans le repertoire temporaire de calcul : $LOCAL_WORK_DIR
cd $LOCAL_WORK_DIR

# Execution du programme gaussian
setenv PATH "$PATH:.:/soft/g03c02/g03"
setenv g03root /soft/g03c02
setenv GAUSS_SCRDIR $LOCAL_WORK_DIR
setenv GAUSS_EXEDIR /soft/g03c02/g03
setenv LD_LIBRARY64_PATH /soft/g03c02/g03
g03 < gaussian.in > $LOCAL_SPOOL_DIR/gaussian.log

# On deplace les fichiers dans le repertoire temporaire de rapatriement des donnees
mv gaussian.chk $LOCAL_SPOOL_DIR
```


Environnement de compilation XLF

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 : utiliser des compilateurs *threadsafes* (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, P-threads, ...	xlf_r	xlf90_r	xlf95_r

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

Options : entrée/sortie du compilateur

- -qfixed=132 : fichier source au format fixe
- -qfree=f90 fichier source au format libre
- -qsuffix=f=f90 : extension des fichiers source
- -I*dir* : chemin pour les fichiers à inclure
- -qmoddir=*dir* : répertoire de création des modules

Options : portage

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

Options : déboguage

- -qnooptimize : pas d'optimisation du code
- -qcheck : vérification des accès aux éléments de tableaux
- -qdbg : informations pour le débogueur symbolique
- -qfullpath : inclusion du chemin absolu UNIX
- -qextchk : vérification de la cohérencesdes commons, appels aux sous-programmes
- -qflttrap=ov:und:zero:inv:en : type d'exceptions flottantes tracées
- -qinitauto=FF

Options : optimisations

- -O2, -O3 : niveau d'optimisation
- -qstrict : respect de la sémantique des programmes
- -qhot : optimisations plus agressives
- -qarch=pwr[5|4] : optimisations liées à l'architecture
- -qtune=pwr[5|4] : optimisation du jeu d'instructions
- -qunroll=[auto|yes] : active ou renforce l'analyse des boucles

Options : analyse, profilage, parallélisation

- -Q<x> : inlining (-Q) ou non (-Q!), sélectif dans un même fichier
- -qipa : analyse inter-procédurale globale
  - ➡ -qipa=inline=<x>:inline=[no]auto
- -pg : profilage de l'application
  - ➡ -pg -qdbg -qfullpath ensemble
- -qsmp=omp : reconnaissance des directives OpenMP
- -qreport=[hotlist|smplist] : listings des transformations

Compilation : propositions d'options

- Déboguage :
  - ➡ -qnooptimize -qcheck -qdbg -qfullpath -qflttrap=ov:und:zero:inv:en -qinitauto=FF
- Optimisation :
  - ➡ -qarch=pwr[5|4] -qtune=pwr[5|4] [-qhot] -O3 [-qstrict] [-qsmp=omp] [-qipa]

Edition des liens :

- Compilation en mode 32 bits (le défaut) :
  - ➡ -bmaxdata:0x40000000 : 4 segments de 256 Mo pour les données (heap)
  - ➡ -bmaxstack:0x10000000 : 256 Mo pour les variables locales (stack)
  - ➡ Options INutiles en mode 64 bits
- -brename:.name,.name_ : ajout du caractère '_' à la fin
- -blpdata : demande de LP à l'exécution ; par défaut
  - ➡ Attention aux binaires importés : ldedit -blpdata <executable>
- -pg / -qipa : si elles sont présentes à la compilation

Bibliothèques logicielles

Calcul scientifique (1/2)

- ESSL : algèbre linéaire, calcul matriciel, FFT, valeurs propres, ...
  - ➡ -lessl
- P-ESSL : sous-ensemble parallèle de ESSL
  - ➡ -lpessl
- MASS : fonctions mathématiques usuelles optimisées
  - ➡ -lmass
- MASSv : fonctions mathématiques usuelles traitant des vecteurs
  - ➡ -lmassv

Calcul scientifique (2/2)

- LAPack
  - ➡ -L /soft/library/lapack.[32|64]_pwr[5|4] -llapack
- BLACS
  - ➡ -L /soft/library/lapack.[32|64]_pwr[5|4] -lblacs -lblacsF77init
- ScaLAPack (repose sur BLACS)
  - ➡ -L /soft/library/lapack.[32|64]_pwr[5|4] -lscalapack
- FFTW
  - ➡ -L /soft/library/fftw-3.1.2.[32|64]_pwr[5|4]/lib -lfftw3

Gestion des données

- netcdf 3.6.0
  - ➡ -L/soft/library/netcdf.[32|64] -lnetcdf
- HDF 5-1.6.5, 4.2r1
  - ➡ -L/soft/library/HDF5.[32|64]/[parallel|serial] -lhdf5
- szip
  - ➡ -L/soft/library/szip2.0-aix[64]-enc/lib -lsz
- zlib
  - ➡ -L/soft/library/zlib.[32|64]/lib -lz_[32|64]
- mdsplus
  - ➡ -L/soft/library/mdsplus/lib -lmdsplus

Outils d'aide à l'analyse

Débogueurs

- dbx : débogueur symbolique IBM
- pdbx : version parallèle de dbx
- gdb / ddd : débogueur GNU et interface graphique
- compilation avec les options suivantes recommandée :

  ➡ -qnooptimize -qdbg -qfullpath

Profilage

- gprof : localisation de la consommation cpu
- Marche à suivre :
  - ➡ compilation avec les options -pg -qdbg -qfullpath
  - ➡ exécution du programme
  - ➡ analyse du (des) fichier (s) gmon.out
- Fonctionnement :
  - ➡ gprof a.out gmon.out > analyse_gprof
  - ➡ gprof a.out gmon.0.out [gmon.1.out] ... > analyse_gprof_all
- xprofiler : interface graphique d'analyse
  - ➡ plus convivial que gprof
  - ➡ des fonctionnalités supplémentaires (accès au code source, etc.)

gprof : répartition du temps cpu consommé

```
   cumulative  self         self   total &
time seconds  seconds callsms/callms/callname
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]
```

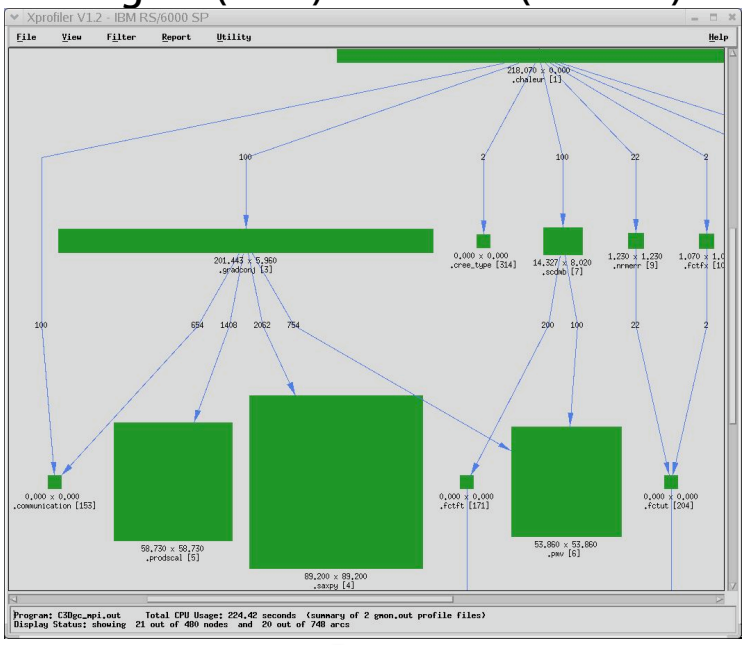
- Classement par **ordre décroissant** du cpu consommé
- La répartition du nombre d'appels peut être trouvée en analysant la hiérarchie des appels de fonction

gprof : hiérarchie des appels

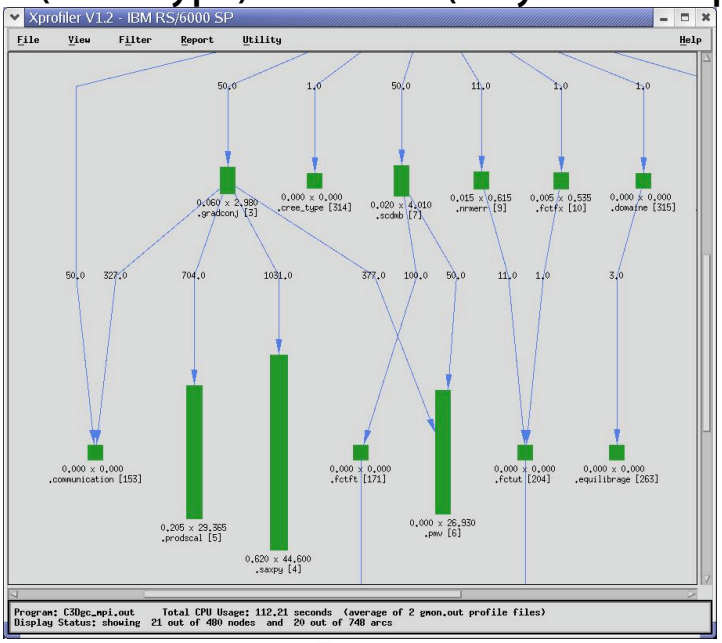
```
                                     called/total  parents
index    %time  self   descendents   called+self  name  index
                                     called/total  children

                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 au-dessous dans le tableau

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

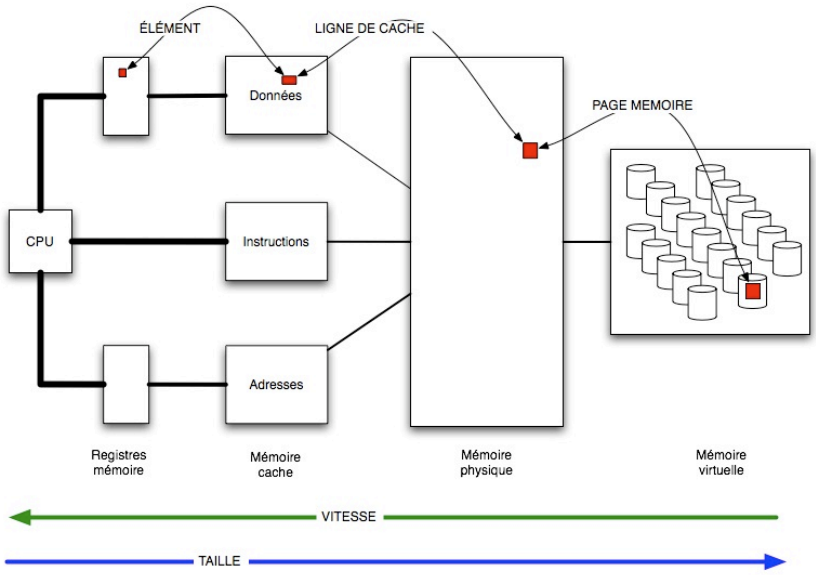


xprofiler : mode average
largeur (écart-type) x hauteur (moyenne des procs.)

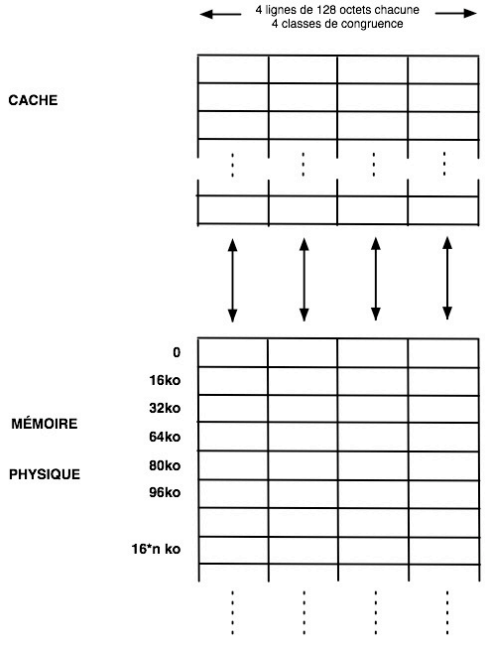


Considérations matérielles

Hiérarchie mémoire



Associativité des caches



Défauts de cache

- Lorsqu'une zone mémoire est accédée, les 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, *cache miss*
  - ➡ il faut aller la cherche au delà, en mémoire
  - ➡ cette perte de temps pénalise l'exécution du programme
- Rechercher une donnée signifie d'abord parcourir une table rapide d'indexation des pages mémoire (TLB)
  - ➡ elle contient les chemins vers les dernières pages mémoire accédées
  - ➡ si la donnée n'est pas dans une de ses pages, il y a un défaut de TLB, *TLB miss*
  - ➡ il faut calculer l'adresse mémoire de la bonne page puis chercher la donnée
  - ➡ l'exécution du programme est pénalisée

Usage efficace de la mémoire

- Tenir compte de l'usage des données par les algorithmes
  - ➡ construire ses tableaux multi-dimensionnels dans l'ordre de leur parcours
- Réduire les défauts de cache : parcourir avec un stride de 1
  - ➡ parcourir séquentiellement les données, bien imbriquer les boucles
- Eviter les conflits d'associativité
  - ➡ éviter les grandes puissances de 2 dans les premières dimensions
- Réduire les défauts de TLB
  - ➡ éviter les strides trop grands

Optimisations scalaires

Préambule

- Ecriture du programme
  - ➡ réflexion sur papier ; algorithmes performants ; déclarations explicites des variables ; commentaires, ...
- Modularité
  - ➡ un sous-programme / module par fichier ; bibliothèques thématiques ; fichier de compilation *makefile* ; ...
- Bibliothèques scientifiques
  - ➡ [P-]ESSL, FFTW, [Sca-]LaPack, MASS[v], ...
- Validation numérique
  - ➡ mode déboguage ; test représentatif avec résultats connus
- Analyse des performances et optimisation

Opérations en virgule flottante

- L'opération de base des processeurs Power est : A + B * X
  - ➡ addition / soustraction seule avec X = ±1
  - ➡ multiplication seule avec A = 0
- La division est beaucoup plus chère que la multiplication
  - ➡ si possible, remplacer les divisions par des multiplications
- La précision des données est limitée à la mantisse
  - ➡ environ 15 décimales en double précision
- Les test arithmétiques doivent être écrits correctement
  - ➡ IF (abs (a-b) < petit_machine_courante) THEN ...
- Les constantes numériques doivent être bien évaluées
  - ➡ pi = ACOS (-1.0_rp) ; sq2 = SQRT (2.0_rp) ; rp = KIND (1.0D0)

Arithmétique entière

- Le parent pauvre du calcul ...
- L'ordre des opérations influe sur le résultat obtenu !
  - ➡ i * { (i-1) / 2 } ≠ { i * (i-1) } / 2 : ce n'est pas associatif !
- La division entière est extrêmement chère
- Se passer des opérations sur les puissances de 2
  - ➡ ir * 8 = ISHFT (ir, 3) ; (i * j * k) / 32 = ISHFT (i * j * k, -5)
  - ➡ Attention : cela ne fonctionne que sur les entiers positifis !
- Les puissances entières doivent être écrites comme des entiers
  - ➡ x = y**2.0 = exp {2.0*ln (y) } très chère à évaluer en comparaison à x = y*y

Variables et expressions

- Utiliser des variables automatiques / locales autant que possible
  - ➡ analyse plus fine par le compilateur
- Privilégier l'allocation dynamique pour les gros tableaux
- Ecrire des expressions identiques en spécifiant les variables dans le même ordre
  - ➡ x = a+b+c+d est différent pour le compilateur de y = a+c+b+d avec -qstrict
- Eviter des types de données de taille particulière
  - ➡ INTEGER*1, REAL*16, ...

Optimisation des boucles (1/7)

- Les boucles de par leur nature représentent l'essentiel de la consommation cpu
- Le compilateur les optimise bien lorsque leur contenu n'est pas trop important
- Le parcours des tableaux doit y être efficace (stride petit !)

++	+	-	--

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

Optimisation des boucles (2/7)

- Fusion (/ fission) des boucles
  - ➡ Cela consiste à regrouper les instructions de plusieurs boucles en une seule

Avantages :

- ➡ meilleur recouvrement des instructions
- ➡ réduction du coût de gestion de boucles

Inconvénient :

- ➡ risque d'accroissement du nombre de défauts de cache

Version non optimisée	Version optimisée

DO I = 1, N
  X = X * A(I) + B(I)
END DO
DO J = 1, N
  Y = Y * A(J) + C(J)
END DO
	DO I = 1, N
	  X = X * A(I) + B(I)
	  Y = Y * A(I) + C(I)
	END DO

Optimisation des boucles (3/7)

- Traitement des tests conditionnels invariants

Version non optimisée	Version optimisée

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

Optimisation des boucles (4/7)

- Traiter les premières / dernières itérations

Version non optimisée	Version optimisée

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

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 !

Optimisation des boucles (5/7)

- Sortir des boucles les appels aux sous-programmes
  - ➡ transmettre les tableaux en argument plutôt que leurs éléments
  - ➡ limiter le nombre d'appels aux fonctions intrinsèques

Version non optimisée	Version optimisée

DO j = 1, n
  DO i = 1, n
  A(i,j) = B(i,j) * SIN (X(i))
  END DO
END DO
 	
ajouter -WF,-DMASSV à la compilation
	#ifdef MASSV
	call sinv (SINX, X, n)
	#else
	DO i = 1, n
	  SINX(i) = SIN (X(i))
	END DO
	#endif
	DO j = 1, n
	  DO i = 1, n
	  A(i,j) = B(i,j) * SINX(i)
	  END DO
	END DO

Optimisation des boucles (6/7)

- Fusionner les tableaux pour augmenter le débit mémoire
  - ➡ chaque processeur dispose de 8 streams qui lui permettent de créer des flux de données

La boucle suivante ...	... peut être remplacée par ...
REAL(rp), DIMENSION(n) :: A1, A2, A3, A4, A5 REAL(rp), DIMENSION(n) :: B1, B2, B3, B4, B5 REAL(rp), DIMENSION(n) :: C1, C2, C3, C4, C5 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	REAL(rp), DIMENSION(5,n) :: A REAL(rp), DIMENSION(5,n) :: B REAL(rp), DIMENSION(5,n) :: C 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.
Si plus de 8 flux sont nécessaires, essayez de fractionner la boucle !

Optimisation des boucles (7/7)

- Déroulage des boucles : effectuer plusieurs itérations d'une même boucle en une seule

- Avantages :
  - ➡ plus d'opportunités pour le processeur
  - ➡ plus de ré-utilisation des données
  - ➡ meilleure exploitation des lignes de cache

- Inconvénients :
  - ➡ plus de registres nécessaires simultanément
  - ➡ code supplémentaire pour traiter les itérations restantes

```
do i = 1, n-mod(n,unroll), unroll      ...   end do
do i = n-mod(n,unroll)+1, n, 1         ...   end do
```

Optimisation des boucles (7/7)

```
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

FMA Floating Point Multiply and Add

Ratio: 8 loads / 4 FMA → 5 loads / 4 FMA

s0, s1, s2, s3 :
scalaires temporaires importants pour aider le compilateur et minimiser les MaJ 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
```


Introduction au calcul parallèle

Quelques définitions ...

- Qu'est-ce que le calcul parallèle ?
  - ➡ C'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, réseau, ...
  - ➡ Techniques logicielles : compilateurs, langage parallèle , bibliothèques de passage de messages, scientifiques, ...
- Pourquoi faire du calcul parallèle ?
  - ➡ 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 les besoins en ressources sur plusieurs processeurs, voire nœuds, voire calculateurs et ainsi disposer de plus de ressources matérielles (notamment la mémoire)

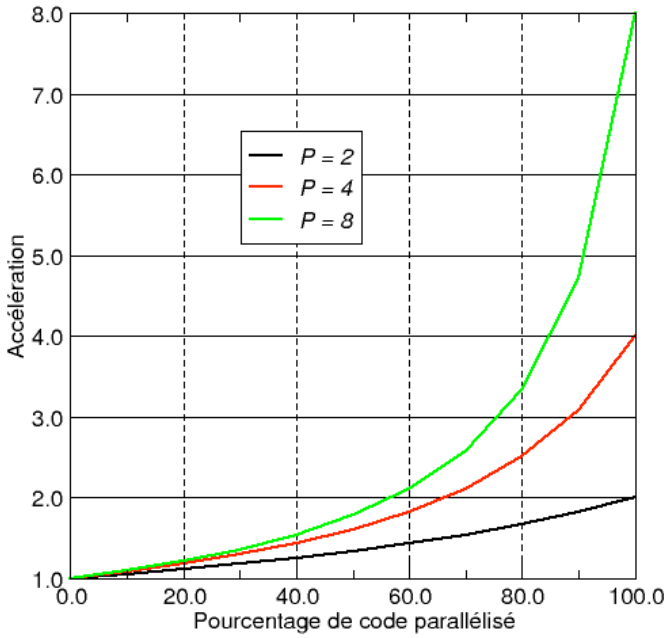
Accélération et Efficacité

- Accélération (speedup) : A(p) = T(1) / T(p)
- Efficacité (Efficency) : E(p) = A(p) / p
  - ➡ N.B. : T(p) est le temps d'exécution sur p processus
    T(1) est le temps du meilleur algorithme séquentiel
- Loi d'Amdhal : gain possible de la parallélisation :

```
A(par,seq) = 1 / ( seq + par/ p) ; par + seq = 1
 seq : portion séquentielle
 par : portion parallèle
```

  - ➡ Exemple : seq = 0.10, alors par= 0.9 et A(par,seq) < 1/seq = 10

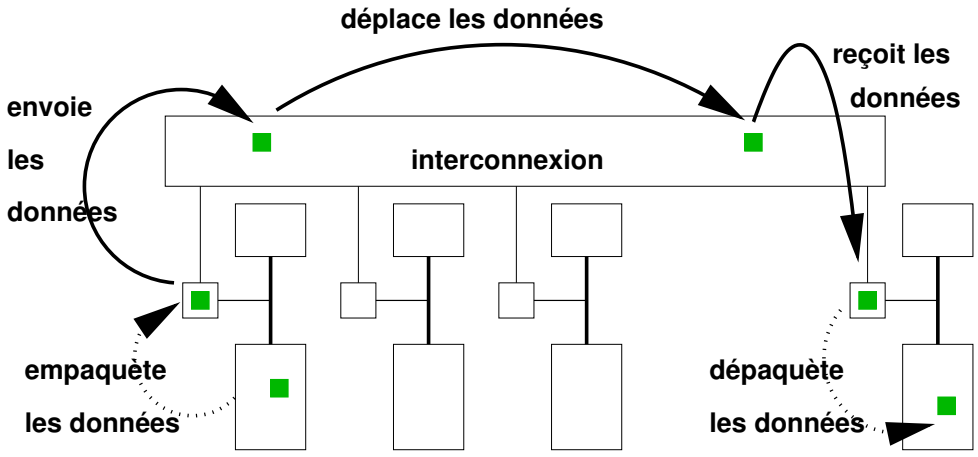
Loi d'Amdahl



Architectures parallèles

Ordinateurs à mémoire distribuée

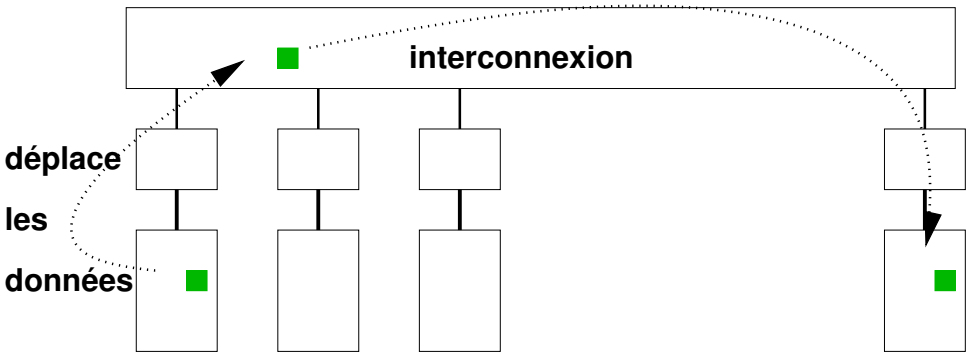
➡ Le système est programmé en utilisant l'échange de messages



Architectures parallèles

Ordinateurs à mémoire partagée

➡Le mouvement des données est transparent pour l'utilisateur



Parallélisation : passage de messages

Qu'est-ce que le passage de messages ?

- Les processus d'une application parallèle se synchronisent, échangent des données, effectuent des opérations globales en envoyant des informations les uns aux autres
- La gestion de ces échanges est réalisée par MPI (Message Passing Interface)
- Explicite, cette technique est à la charge du développeur
- Chaque processus dispose de ses propres données, sans accès direct à celles des autres
- Les processus sont identifiés par leur rang, au sein du groupe

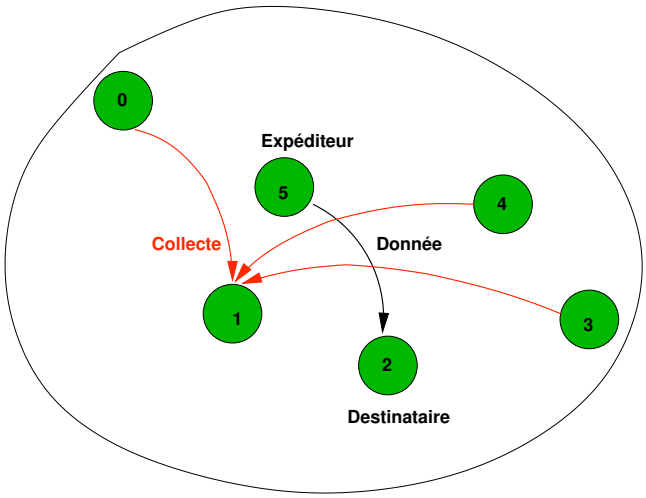
Environnement MPI

- Initialisation au début (`MPI_INIT`)
- Finalisation à la fin (`MPI_FINALIZE`)

```
INTEGER  ::  nbprocs,myrank, 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 des informations au travers de messages, i.e. des communications impliquant au moins deux processus

Analogie avec le courrier électronique



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`
    ligne ou bloc matriciel
  - ➡ données distantes d'un pas variable : `MPI_TYPE_INDEXED`
    triangle dans une matrice
- Type hétérogène :
  - ➡ construction d'une structure : `MPI_TYPE_STRUCT`

Communications point à point

Elle se fait entre deux processus : expéditeur et destinataire

Elle comporte :

- le communicateur, i.e. l'espace de communication (`comm`)
- les deux identifiants (`src`, `dest`)
- la donnée (`buf`), son type (`datatype`) et sa taille (`count`)
- une étiquette (`tag`) qui permet au programme de distinguer les messages

Communications bloquantes et non bloquantes

```
CALL MPI_SEND  (buf, count, datatype, dest, tag, comm, ierr)
CALL MPI_RECV  (buf, count, datatype, src , tag, comm, status, ierr)
CALL MPI_ISEND (buf, count, datatype, dest, tag, comm, irq, ierr)
CALL MPI_IRECV (buf, count, datatype, src , tag, comm, irq, ierr)
```


Communications collectives (1/2)

- La communication collective est une communication qui implique un ensemble de processus qui l'effectuent tous

- Les synchronisations globales :

  ➡ barrière de synchronisation sur l'ensemble des membres du communicateur

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

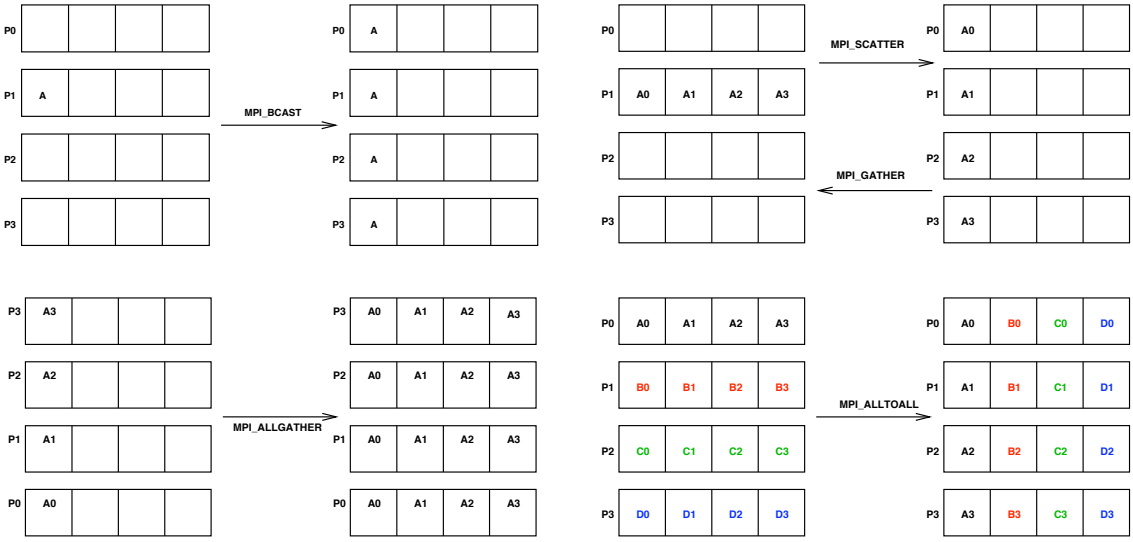
- Les opérations de réduction sur des données réparties

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

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

Communications collectives (2/2)

- Les opérations de diffusion / collecte

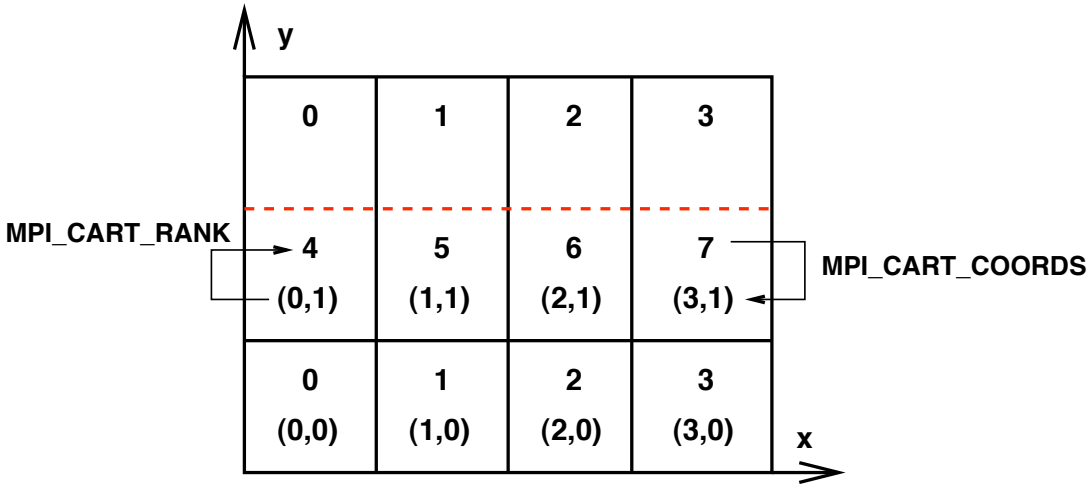


Topologie (1/2)

- Exemple de 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` / `MPI_CART_RANK`

Topologie (2/2)

Exemple de grille 2D, périodique en y



MPI-2 : extension de la norme

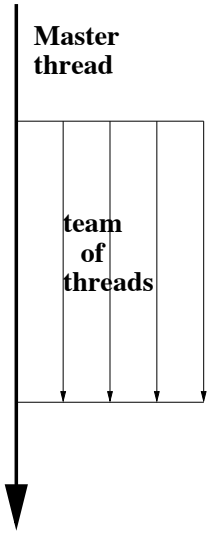
- Il manque un certain nombre de fonctionnalités :
  - ➡ gestion dynamique des processus, (apport majeur mais complexe)
  - ➡ interfaçage avec le fortran 95 et le C++,
  - ➡ entrées / sorties parallèles, (apport majeur)
  - ➡ communications de mémoire à mémoire.
- MPI-2 offre ces fonctionnalités, pas forcément implémentées
  - ➡ MPICH2 (http://www-unix.mcs.anl.gov/mpi/mpich2) (complète)
  - ➡ open-mpi (http://www.open-mpi.org)
  - ➡ Fujitsu, NEC, SUN (complètes)
  - ➡ N.B. : L'implémentation d' IBM n'est pas complète

Parallélisation : mémoire partagée

Principe d'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 fort & join :
  - ➡ Une équipe de threads est créée à l'entrée d'une région parallèle
  - ➡ Son effectif est contrôlé par variable d'environnement ou appel à une librairie
  - ➡ Avant et après, l'exécution est séquentielle
  - ➡ Les threads sont identifiées par leur rang et celui de la thread maîtresse est 0



Directives de compilation

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

```
!$OMP PARALLEL [CLAUSE [, CLAUSE] ... ]
nbthd = 1
!$ nbthd = OMP_GET_NUM_THREADS ()
```

où :

`!$OMP` ou `C$OMP` est une sentinelle, obligatoire comme préfixe

`!$` ou `C$` permet d'effectuer une compilation conditionnelle

`PARALLEL` est une directive

`[CLAUSE [, CLAUSE] ... ]` sont les clauses, optionnelles, pour décrire la visibilité des variables

Variables privées

- Les variables privées (clause `PRIVATE`) ont une adresse unique pour chaque thread : elles sont dupliquées
- Elles ne sont pas accessibles en dehors de la région parallèle
- Les constantes (`PARAMETER`), les arguments en entrée (`INTENT(IN)`) sont privées
- Exemples :

  indices de boucle, variables scalaires (résultats intermédiaires), variables locales, tableaux automatiques, ...

Variables publiques

- Par défaut, dans les constructions parallèles, toutes les variables sont publiques (clause `SHARED`)

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

- Les gros tableaux de données sur lesquels les threads travaillent (de manière disjointe), devraient être `SHARED`

- Exemples :

  variables locales rémanentes (`DATA`,`SAVE`), celles déclarées dans les modules, ou passées en arguments)

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`)

- La variable de réduction doit être partagée entre les threads

- La norme OpenMP 2.0 étend la réduction aux tableaux

- Exemples :

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

Constructions work sharing (1/2)

- Elles permettent le partage de travail entre les threads de manière automatique

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

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

- Une région parallèle peut se limiter à une construction

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


Constructions work sharing (2/2)

Boucle dans une région parallèle

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

Construction parallèle restreinte à une double boucle

```
!$OMP PARALLEL DO SHARED (nx,ny,x,y,z) PRIVATE (i,j)
   DO j = 1, ny
      DO i = 1, nx
         x(i,j) = y(i,j) + z(i,j)
      END DO
   END DO
!$OMP END PARALLEL DO
```

Sérialisation et synchronisation

- Plusieurs constructions permettent de spécifier l'ordre d'accès à des données partagées
  - ➡ section `MASTER ... END MASTER` : pour la thread de rang 0 uniquement
  - ➡ section `SINGLE ... END SINGLE` : pour une seule thread, la première arrivée
  - ➡ section `CRITICAL ... END CRITICAL` : toutes, mais une seule thread à la fois
  - ➡ directive `ATOMIC` : section CRITICAL formée d'une seule instruction
  - ➡ directive BARRIER : barrière de synchronisation globale
- Remarques :
  - ➡ La section `MASTER ...END MASTER` n'a pas de synchronisation implicite à la fin contrairement à `SINGLE ... END SINGLE`
  - ➡ Elle coûte aussi un peu moins cher
  - ➡ La directive `ATOMIC` est moins chère que la section `CRITICAL...END CRITICAL`

Exemple de parallélisation : Équation de la Chaleur 2D

Définition du problème

Les équations ...

$$\begin{cases} \dfrac{\partial u}{\partial t} - \lambda \Delta u = f \text{ dans } \Omega, & \text{avec } \Omega = [0,1]^2 \\ u = 0 \text{ sur } \Gamma = \partial\Omega \end{cases}$$

La parallélisation est faite par :

- décomposition de domaine avec MPI,
- partage du travail avec OpenMP

La discrétisation spatiale est faite par un schéma aux différences finies d'ordre 2

La discrétisation temporelle est faite par un schéma semi-implicite d'ordre 2 :

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

Problème discrétisé

Le laplacien est ainsi discrétisé

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

Le système global s'écrit alors

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

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

Algorithme séquentiel

1. Initialisations

➡ initialisations des vecteurs et des constantes

2. Boucle en temps

2.1. Construction du second membre

2.2. Résolution du système linéaire : **gradient conjugué**

➡ produit matrice-vecteur (pmv)

➡ produit scalaire (prodscal)

➡ combinaison linaire de vecteurs (saxpy)

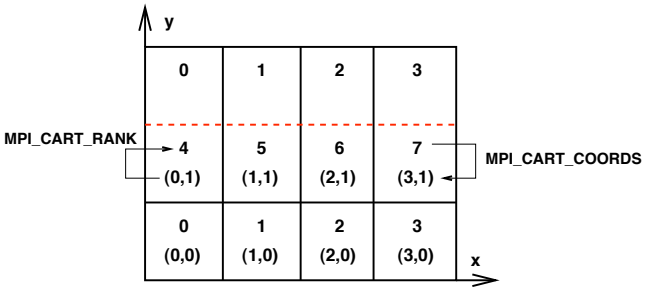
2.3. Avancement en temps

3. Finalisations

Décomposition de domaine

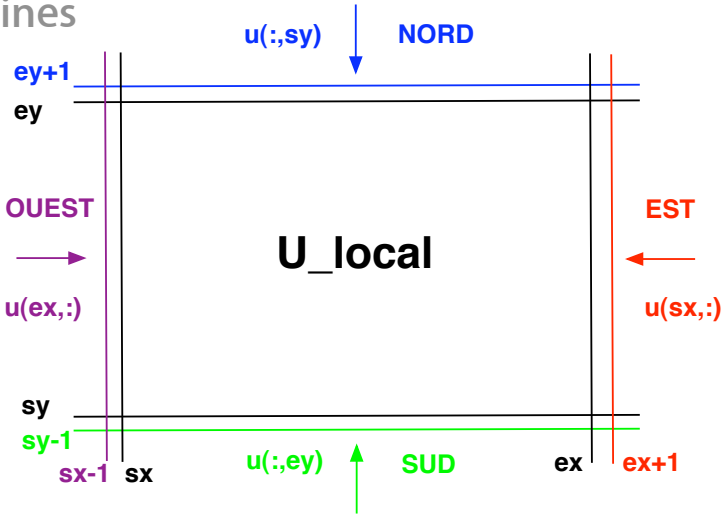
- Partage du domaine en somme directe des sous-domaines
- Parallélisation explicite (à la charge du développeur)
- Pour maillages structurés et non structurés (nœuds, ...)
- Grille de processus sur grille des sous-domaines

➡ topologie MPI



Cellules fantômes

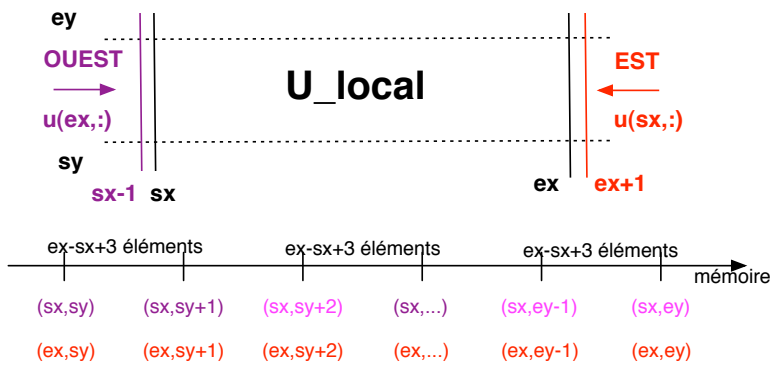
- Nœuds ou cellules fantômes conçus pour stocker des valeurs extérieures nécessaires au calcul des quantités sur les bords des sous-domaines
- Mise à jour de ces cellules à chaque pas de temps
- Les sous-domaines sont des rectangles
  - ➡ les zones d'interface sont des segments



Types de données pour les messages (1/2)

- Dans la direction X, il s'agit de (ey-sy+1) points non contigus mais équidistants (écart = ex-sx+3 !)

CALL MPI_TYPE_VECTEUR (ey-sy+1, 1, ex-sx+3, ITYPE_REAL, colonne, ierr)
CALL MPI_COMMIT (colonne, ierr)

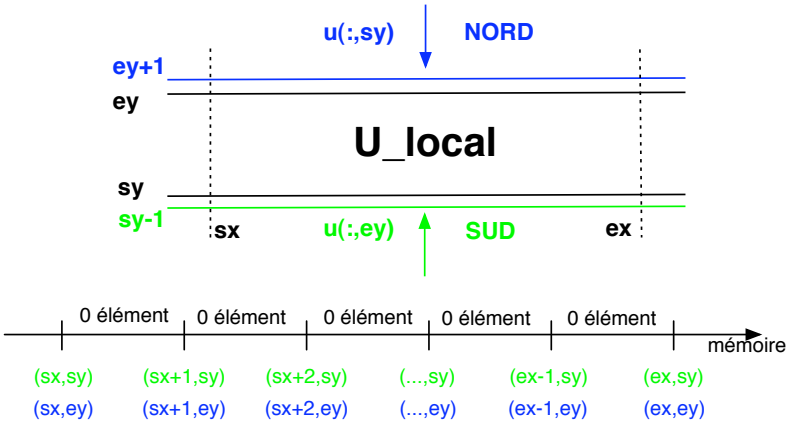


Types de données pour les messages (2/2)

- Dans la direction Y, il s'agit de (ex-sx+1) points contigus

  CALL MPI_TYPE_CONTIGUOUS (ex-sx+1, ITYPE_REAL, ligne, ierr)

  CALL MPI_COMMIT (ligne, ierr)



Communications

- Synchronisation des processus
  - ➡ phase de calcul
  - ➡ phase de communication
- Echange entre les processus, 2 à 2 : `MPI_SENDRECV`
  - ➡ `MPI_SEND` et `MPI_RECV` en une seule communication

```
CALL MPI_SENDRECV (a(sx,ey), 1, ligne, voisin(N), tag1, a(sx,sy-1), 1,
ligne, voisin(S), comm2d, status, ierr)

CALL MPI_SENDRECV (a(sx,sy), 1, ligne, voisin(S), tag1, a(sx,ey+1), 1,
ligne, voisin(N), comm2d, status, ierr)
```

➡ De même pour l'autre côté

Algorithme parallèle MPI

1. Initialisations
   - ➡ initialisations des vecteurs et des constantes
   - ➡ construction de la topologie et des types MPI
2. Boucle en temps

2.1. Construction du second membre

2.2. Résolution du système linéaire : **gradient conjugué**
   - ➡ produit matrice-vecteur (pmv)
   - ➡ produit scalaire (prodscal)
   - ➡ combinaison linaire de vecteurs (saxpy)
   - ➡ communications / synchronisations

2.3. Avancement en temps

3. Finalisations
   - ➡ destruction de la topologie et des types MPI

Partage du travail

- Distribution des itérations entre les processus
- Parallélisation implicite
- Après analyse, les boucles sont parallélisées ou non :
  - ➡ la boucle en temps : dépendance, elle n'est pas parallélisable
  - ➡ la boucle du GC : dépendance, elle n'est pas parallélisable
  - ➡ les boucles produit matrice-vecteur : parallélisables
  - ➡ le produit scalaire : réduction donc parallélisable
  - ➡ combinaison linéaire de vecteurs : parallélisable
- Création d'une région parallèle qui englobe la boucle en temps
  - ➡ toutes les threads l'effectuent mais elles se partagent celles parallélisables

Traitement de la 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    <--- variables communes
   tps = tps + dt            <--- à l'ensemble des threads
   itgc = 0                  <--- donc partagées
   rnorm_k = zero            <--- et mises à jour une seule fois
!$OMP END MASTER
   CALL  gradconj (u, b, p, q, r, it_max, prec, itgc, rnorm_k, rnudt2)
END DO
!$OMP END PARALLEL
```


Boucle de convergence du CG (1/2)

```
CALL pmv      (r, u, - rnudt2)
CALL saxpy    (r, -one, b, one)
CALL prodscal (rnorm_k, r, r)   <--- argument en sortie, donc à partager
it = 0   <--- variables locales, donc privées
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)
!$OMP MASTER
   itgc = it   <--- argument en sortie, donc à partager
   alpha_k = rnorm_k / alpha_k   <--- variables locales initialisées
   beta_k  = rnorm_k                  car à partager
   rnorm_k = 0.0_rp              <--- argument en sortie, donc à partager
!$OMP END MASTER
!$OMP BARRIER
...
```

Boucle de convergence du CG (2/2)

```
...
   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                 <--- attention à la mise à jour
!$OMP BARRIER
   CALL saxpy( p, beta_k, r, one )
   convergence = ( SQRT(rnorm_k) < prec )
        ^--- variable locale, donc privée
END DO
```


Boucles parallélisées : pmv (1/2)

```
SUBROUTINE pmv (a, b)

   c1 = scal / hx**2   <--- variable locale, donc privée
   c2 = scal / hy**2   <--- variable locale, donc privée
   c3 = one - 2.0_rp * (c1 + c2)

$OMP DO
   DO j = sy, ey     <--- variables transmises par module
      DO i = sx, ex       donc partagées
         a(i,j) = c3 * b(i,j)
                + c1 * ( b(i+1,j  ) + b(i-1,j  ) )
                + c2 * ( b(i  ,j+1) + b(i  ,j-1) )
      END DO              ^     variables transmises par argument
   END DO                 |--   donc partagées
$OMP END DO
```

Boucles parallélisées (2/2) : prodscal et saxpy

```
SUBROUTINE prodscal (rnorm, a, b)
!$OMP DO REDUCTION( +:rnorm )
   DO j = sy, ey    <--- variables transmises par module
      DO i = sx, ex      donc partagées
         rnorm = rnorm + a(i,j) * b(i,j)
      END DO       ^    argument en sortie, donc partagé,
   END DO          |--- donc réduction
!$OMP END DO


SUBROUTINE saxpy (a, scala, b, scalb)
!$OMP DO
   DO j = sy, ey    <--- variables passées par module
      DO i = sx, ex      donc partagées
         a(i,j) = scala * a(i,j) + scalb * b(i,j)
      END DO        ^     ^          ^     ^   variables transmises par
   END DO           |-----|----------|-----|-- argument donc partagées
!$OMP END DO
```


CRIHAN

Centre de Ressources Informatiques de Haute-Normandie