École Polytechnique

Calcul Parallèle

Eric Goubault — Commissariat à l'Energie Atomique
Frédéric Nataf, Marc Schoenauer — Ecole Polytechnique

Calcul Parallèle

Eric Goubault, Frédéric Nataf, Marc Schoenauer

14 janvier 2000

Table des matières

Préface

Ce cours a été originellement défini et enseigné par Patrick Cousot (Ecole Normale Supérieure) et Alain Bamberger (Institut Français du Pétrole). Les notes qui suivent se sont grandement inspirées de leurs cours.

L'objectif est de montrer à l'étudiant comment programmer efficacement une machine parallèle, à travers un certain nombre d'exemples venant du monde du calcul scientifique. En passant, on en profite pour parler d'algorithmique parallèle et distribuée, d'architecture, de sémantique, et bien entendu de mathématiques appliquées.

On a choisi comme support informatique, la librairie de passage de messages PVM, qui permet de programmer aussi bien des super-calculateurs que des réseaux de stations. Cette librairie est très utilisée dans le monde scientifique (même si elle est maintenant détronée par MPI) et ressemble fortement à la plupart des autres librairies de passages de messages utilisées par ailleurs. Nous faisons un comparatif à l'annexe B. Le "kit de survie" pour programmer (sous PVM) est principalement constitué des chapitres 1 et 2, ainsi que de l'annexe C. Des compléments sont faits dans les chapitres 5 et 6, mais ne sont tout de même pas exhaustifs; pour plus de détails on se reportera au livre [GBD+94].

Les notions plus pérennes du parallélisme (avec une idée d'application scientifique) sont développées au chapitre 7. Son but principal est de faire comprendre la difficulté à assurer un bon rendement et de bonnes performances sur des machines parallèles, à travers des exemples. Les notions pérennes du parallélisme (coté sémantique, c'est à dire avec l'idée d'assurer la correction du programme plutôt que sa performance) sont brièvement développées à l'annexe A.

Enfin, deux applications typiques du calcul scientifiques (parallèle) sont entièrement décrites dans les chapitres 3, 4 et 8. Les deux premiers sont consacrés à des problèmes directs alors que le dernier est plutôt consacré à des problèmes inverses. On discute non seulement les mathématiques sous-jacentes mais également l'implémentation et la parallélisation du calcul.

Un complément utile du cours est le polycopié d'Alain Bamberger [Bam98]. On pourra également consulter les compléments et errata du cours à l'adresse `http://w3.edu.polytechnique.edu/profs/informatique/Eric.Goubault` ainsi que la page web de l'ORAP `http://www.irisa.fr/orap/`, de PVM `http://www.epm.ornl.gov/pvm/` et de MPI `http://www.mcs.anl.gov/mpi/index.html`.

Enfin, on conseille la lecture (même rapide) de livres tels que [LER92] et [CS98]. Des cours similaires ou complémentaires ont été enseignés par ailleurs, en particulier à l'IDRIS `http://www.idris.fr/data/cours/cours-IDRIS.html#PVM` au MIT `http://command.mit.edu/18.337` et à Berkeley `http://now.CS.Berkeley.EDU/cs267/`.

Chapitre 1

Introduction au parallélisme

1.1 Une classification des machines parallèles

Une machine *parallèle* est essentiellement un ensemble de *processeurs* qui *coopèrent* et *communiquent*.

Historiquement, les premières machines parallèles sont des *réseaux* d'ordinateurs, et des machines *vectorielles* et faiblement parallèles (années 70 - IBM 360-90 vectoriel, IRIS 80 triprocesseurs, CRAY 1 vectoriel...).

On distingue classiquement quatre types principaux de parallélisme (Taxonomie de *Tanenbaum*): *SISD*, *SIMD*, *MISD* et *MIMD*. De nos jours cette classification peut paraître un peu artificielle car le moindre micro-processeur courant inclut lui-même plusieurs formes de micro-parallélisme. Elle permet néanmoins d'expliquer les bases de l'architectures des ordinateurs, séquentiels et parallèles. Cette classification est basée sur les notions de flot de *contrôle* (deux premières lettres, I voulant dire "Instruction") et flot de *données* (deux dernières lettres, D voulant dire "Data").

1.1.1 Machine SISD

Une machine *SISD* (*S*ingle *I*nstruction *S*ingle *D*ata) est ce que l'on appelle d'habitude une machine de Von Neuman. Une seule instruction est exécutée et une seule donnée (simple, non-structurée) est traitée à tout instant.

Le code suivant,

```
int A[100];
...
for (i=1;100>i;i++)
   A[i]=A[i]+A[i+1];
```

s'exécute sur une machine séquentielle en faisant les additions `A[1]+A[2]`, `A[2]+A[3]`, etc., `A[99]+A[100]` à la suite les unes des autres.

1.1.2 Machine SIMD

Une machine *SIMD* (*S*ingle *I*nstruction *M*ultiple *D*ata) peut être de plusieurs types, parallèle ou systolique. En général l'exécution en parallèle de la même instruction se fait en même temps sur des processeurs différents (parallélisme de donnée *synchrone*). Examinons par exemple le code suivant écrit en CM-Fortran sur la Connection Machine-5 avec 32 processeurs,

```
      INTEGER I,A(32,1000)
CMF$  LAYOUT A(:NEWS,:SERIAL)
```

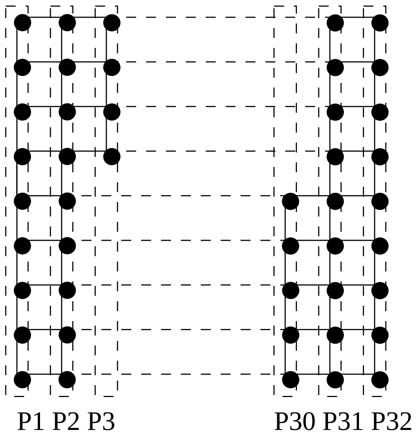


FIG. 1.1 – *Répartion d'un tableau sur les processeurs d'une machine SIMD typique*

```
    ...
    FORALL (I=1:32,J=1:1000)
  $     A(I:I,J:J)=A(I:I,J:J)+A(I:I,(J+1):(J+1))
```

Chaque processeur i, $1 \leq i \leq 32$ a en sa mémoire locale une tranche du tableau `A`: `A(i,1), A(i,2), ..., A(i,1000)`. Il n'y a pas d'interférence dans le calcul de la boucle entre les différentes tranches: tous les processeurs exécutent la même boucle sur leur propre tranche en même temps (cf. figure 1.1).

1.1.3 Machine MISD

Une machine *MISD* (*M*ultiple *I*nstruction *S*ingle *D*ata) peut exécuter plusieurs instructions en même temps sur la même donnée. Cela peut paraître paradoxal mais cela recouvre en fait un type très ordinaire de micro-parallélisme dans les micro-processeurs modernes: les processeurs vectoriels et les architectures pipelines.

Un exemple de "pipelinage" d'une addition vectorielle est le suivant. Considérons le code:

```
FOR  i:=1 to n DO
   R(a+b*i):=A(a'+b'*i)+B(a''+b''*i);
```

`A`, `B` et `R` sont placés dans des registres vectoriels qui se remplissent au fur et à mesure du calcul,

Temps	A	(	i	)	B	(	i	)	R	(	i	)
1	1	.	.	.	1	.	.	.	.	.	.	.
2	2	1	.	.	2	1	.	.	.	.	.	.
3	3	2	1	.	3	2	1	.	.	.	.	.
4	4	3	2	1	4	3	2	1	.	.	.	.
5	5	4	3	2	5	4	3	2	1	.	.	.
6	6	5	4	3	6	5	4	3	2	1	.	.
etc.												

En ce sens, quand le pipeline est rempli, plusieurs instructions sont exécutées sur la même donnée.

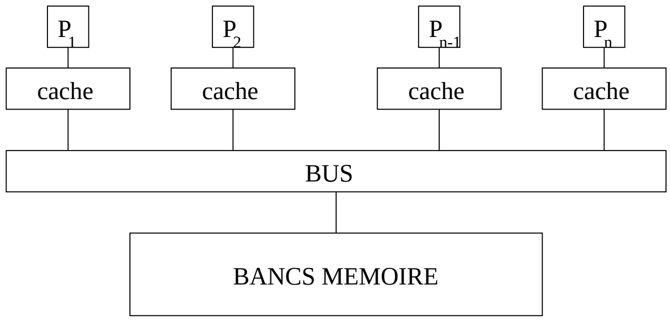


FIG. 1.2 – *Architecture simplifiée d'une machine à mémoire partagée*

1.1.4 Machine MIMD

Le cas des machines *MIMD* (*M*ultiple *I*nstruction *M*ultiple *D*ata) est le plus intuitif et est celui qui va nous intéresser le plus dans ce cours. On a plusieurs types d'architecture possibles:

(1) Mémoire partagée (Sequent)

(2) Mémoire locale avec réseau de communication (Transputer, Connection Machine, local, par réseau d'interconnexion), ou système réparti

C'est le cas (2) que l'on va voir plus particulièrement avec PVM. On pourra également simuler le cas (1) [1].

Mémoire Partagée:

Une machine MIMD à mémoire partagée [2] est principalement constituée de processeurs avec des horloges indépendantes, donc évoluant de façon asynchrone, et communiquant en écrivant et lisant des valeurs dans une seule et même mémoire (la mémoire partagée). Une difficulté supplémentaire, que l'on ne décrira pas plus ici, est que chaque processeur a en général au moins un cache de données (voir figure 1.2), tous ces caches devant avoir des informations cohérentes aux moments cruciaux.

La synchronisation des exécutions des processeurs (ou processus, qui en est une "abstraction logique") est nécessaire dans certains cas. Si elle n'était pas faite, il y aurait un risque d'*incohérence* des données. Partant de $x = 0$, exécutons $x := x + x$ en parallèle avec $x := 1$. On a alors essentiellement trois exécutions possibles (en supposant chacune des affectations compilées en instructions élémentaires insécables, ou "*atomiques*", comme suit):

`LOAD x,R1`	`WRITE x,1`	`LOAD x,R1`
`LOAD x,R2`	`LOAD x,R1`	`WRITE x,1`
`WRITE x,1`	`LOAD x,R2`	`LOAD x,R2`
`WRITE x,R1+R2`	`WRITE x,R1+R2`	WRITE x,R1+R2
Résultat x=0	Résultat x=2	Résultat x=1

Cela n'est évidemment pas très satisfaisant; il faut rajouter des synchronisations pour choisir tel ou tel comportement, et en tout cas éviter le troisième comportement en général. Les deux premiers sont acceptables en un certain sens, ils sont deux possibles *séquentialisations* des processus parallèles.

1. On se reportera à ce propos à la section 7.1.3.
2. C'est ce qui est simulé par exemple par les threads JAVA, sur un seul processeur.

La synchronisation peut se faire par différents mécanismes:

- Barrières de synchronisation,
- Sémaphores : deux opérations P et V.
- Verrou (mutex lock) : sémaphore binaire qui sert à protéger une section critique.
- Moniteurs : construction de haut niveau, verrou implicite.
- etc.

Les opérations P et V sur les sémaphores sont parmi les plus classiques et élémentaires. On peut considérer qu'à chaque variable partagée x dans la mémoire est associé un "verrou" (du même nom) indiquant si un processus est en train de manipuler la variable, en interdisant son accès. L'opération Px exécutée par un processus verrouille ainsi son accès exclusif à x. L'opération Vx ouvre le verrou et permet à d'autres processus de manipuler x à leur tour.

La encore des erreurs sont possibles, en voulant trop synchroniser par exemple. Il peut y avoir des cas d'*interblocage* (deadlock, livelock) en particulier.

Supposons qu'un processus T_1 ait besoin (pour effectuer un calcul donné) de verrouiller puis déverrouiller deux variables x et y dans l'ordre suivant: Px puis Py puis Vx puis Vy alors qu'un autre processus, en parallèle, désire faire la séquence Py, Px puis Vy et enfin Vx. En fait les deux processus peuvent s'interbloquer l'un l'autre si T_1 acquiert x (respectivement T_2 acquiert y) puis attend y (respectivement x).

Machine distribuée

En général, l'emploi d'autres mécanismes de communication que la mémoire partagée pour une machine MIMD est due au fait que les processeurs sont physiquement trop éloignés pour qu'un partage de petites informations soit raisonnable. Par exemple, le réseau Internet permet de considérer en un certain sens, tous les ordinateurs reliés comme étant un seul et même ordinateur distribué, où les processeurs travaillent de façon asynchrone et où les informations transitent par passage de message. Un certain nombre de super-calculateurs travaillent par échange de messages également.

De façon générale, la synchronisation et l'échange d'information peuvent se faire par,

- Appel de procédure distribuée (RPC) :
  - réponse synchrone
  - réponse asynchrone
- Envoi/Réception de message asynchrone (tampon de taille limitée ou non); active polling ou gestion à l'arrivée par une procédure `handler`.
- Envoi/Réception de message synchrone : rendez-vous.
- Mélange des deux derniers cas.

Le protocole RPC ("Remote Procedure Call") entre machines UNIX avec réseau Ethernet est un exemple de programmation MIMD (cf. figure 1.3).

1.1.5 Conclusion

Les architectures sont plus ou moins bien adaptées à certains problèmes. En fait, le gain de temps espéré, qui est d'au plus N (nombre de processeurs) [3] est rarement atteint et, en pratique, parallélisme difficile à contrôler. Enfin, pour revenir à notre

3. Il existe néanmoins des problèmes avec gains moyens supérieurs (recherche distribuée).

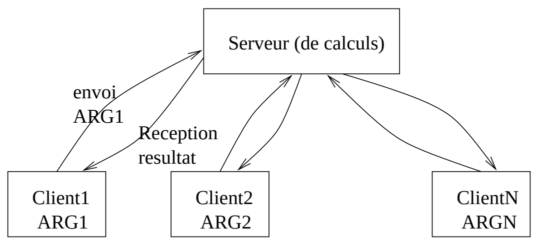


FIG. 1.3 – *Architecture Client/Serveur (RPC)*

première remarque sur la taxonomie de Tanenbaum, il y a de plus en plus rarement une distinction tranchée entre le modèle mémoire partagée et celui par passage de messages; dans la realité, la situation est très hybride.

1.2 Contrôle d'une machine parallèle

Il y a deux méthodes principales,

(1) On dispose d'un language séquentiel: le compilateur *parallélise* (Fortran...),

(2) On a une extension d'un langage séquentiel ou un langage dédié avec des constructions parallèles *explicites* (Parallel C, Occam...)

On verra brièvement les techniques pour le cas (1) à la section 7.2. On se concentre ici principalement sur le deuxième cas. Il faut alors disposer de primitives (ou instructions) pour:

- La création de parallélisme,
  - Itération simultanée sur tous les processeurs (*FOR* parallèle de Fortran ou *FORALL*),
  - Définition d'un ensemble de *processus* (*COBEGIN* cf. figure 1.4),
  - Création de *processus* (*FORK* de Unix)
- Contrôler les tâches parallèles,
  - Synchronisation (*Rendez-vous* d'Ada),
  - Passage de messages (synchrones/asynchrones, *broadcast*,...),
  - Section critiques (gestion de la mémoire partagée),
  - Arrêt d'une exécution parallèle (*COEND*, *WAIT* d'Unix)

1.3 Quelques exemples de machines parallèles

Dans cette section, l'objectif est de donner une idée des super-calculateurs utilisés dans le monde industriel et du calcul scientifique.

1.3.1 La Connection Machine-5

Elle est composée de,

- 32 à 65536 noeuds qui chacun comporte,
  - un processeur Sparc (32 MHz, 22 Mips, 5 MFlops)
  - 4 unités vectorielles (32 MFlops chacune) pour les opérations flottantes

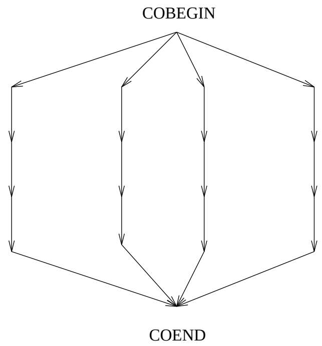


FIG. 1.4 – *Représentation simplifiée du flot d'exécution dans le cas d'un* COBEGIN/COEND

  - 4 bancs mémoire de 8 Mo de DRAM
- Réseau d'interconnexion en arbre gras (hypercube), entre voisins on a: bande passante: 20 Mo/s, latence: 5 μs
- Entrées/Sorties en parallèle à un système de disques, bande passante: 33 Mo/s, latence: 75 ms

Plusieurs modèles (voir figure 1.5) ont été installés en France, dont un à l'ETCA (32 noeuds, 2 processeurs de contrôle, 1 Go de RAM, 16 Go de disques, 4 Gflops crête, mai 1993).

Elle peut fonctionner en,

- mode SIMD: CM Fortran (sous-ensemble de Fortran 90 et instructions spécifiques d'organisation des données en parallèle), C*
- mode MIMD: de même mais en utilisant la librairie CMMD de passage de messages (on retrouve des ordres similaires dans PVM)

1.3.2 Cray T3E

La machine la plus puissante chez CRAY, deux types de modèles, refroidissement à air (pas plus de 128 processeurs) ou par un liquide (jusqu'à 2048 processeurs). Pour les deux types de modèles on peut avoir des configurations (voir figures 1.6 et 1.6):

- De 16 à 2048 noeuds comprenant chacun,
  - un DEC Alpha EV5 (600 MFlops)
  - de 64 Mo à 2 Go de mémoire vive
- Un réseau d'interconnexion en tore 3D (bande passante de 2 à 128 Go/s). Entrées sorties par un réseau en double anneau (allant dans des sens inverses), "GigaRing"

Performance de crête de 9.6 GFlops à 1.2 TFlops, jusqu'à 4 To de mémoire vive. Il peut être utilisé,

- langages: Fortran 90, C et C++ (compilateurs optimisants)
- mode MIMD en utilisant la "Cray Message Passing Toolkit" supportant PVM, MPI (passage de messages) ou la "Shared Memory Library" (le CRAY T3E est organisé logiquement en mémoire partagée même si physiquement ce n'est pas le cas)
- mode SIMD en utilisant Fortran 90, CRAFT ou HPF

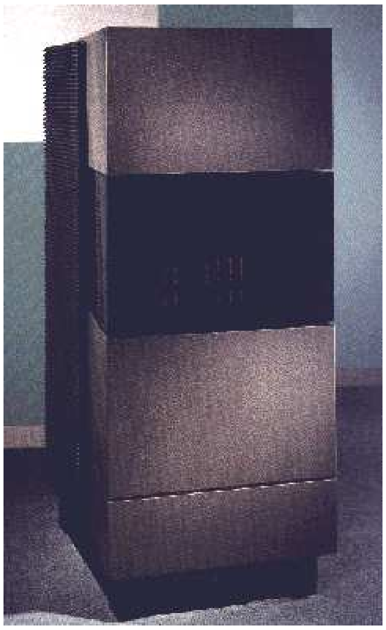


FIG. 1.5 – *Une Connection-Machine 5*

FIG. 1.6 – *Deux Cray T3E*

FIG. 1.7 – *Un IBM SP2*

1.3.3 IBM SP2

L'IBM RISC/6000 Scalable POWERparallel System (SP) est constitué de (voir figure 1.7):

- Noeuds avec un processeur RS/6000 soit à 120 MHz ("thin node" – 480 MFlops) avec 256Mo de mémoire vive et 1,8 Go de mémoire disque, soit à 135 MHz ("wide node" – 540 MFlops) avec 1 à 2 Go de mémoire vive.
- Communication par passage de messages à travers un réseau cross-bar à deux niveaux; bande passante de 150Mo par seconde.
- Fonctionne en MIMD avec PVM ou MPI
- Fonctionne en SIMD avec HPF (High Performance Fortran)

1.3.4 Réseaux de stations

Un réseau de stations fonctionne en mode MIMD avec une communication par passage de messages (typiquement gérés par PVM) sur un réseau local ou d'interconnexion.

En fait, un réseau de stations est souvent une architecture hybride dans le sens où il peut y avoir combinaison sur un réseau local de stations et de machines multiprocesseurs (elles-même gérant leur parallélisme interne par mémoire partagée ou par bus ou réseau). PVM peut également gérer ce genre d'architecture.

1.3.5 Top 500

Il est établi régulièrement une liste des 500 machines les plus puissantes, voir à ce propos `http://www.top500.org`. La puissance est mesurée par un benchmark d'algèbre linéaire, indiquée ci-dessous par le nombre Rmax, qui est la performance en Gflop/s pour le problème le plus gros programmé sur l'ordinateur. En novembre 1999 on trouvait:

Rang	Constructeur	Ordinateur	Rmax
1	Intel	ASCI Red	2379.6
2	IBM	ASCI Blue-Pacific SST, IBM SP 604e	2144
3	SGI	ASCI Blue Mountain	1608
4	Cray/SGI	T3E1200	891.5
5	Hitachi	SR8000/128	873.6
...	...	...	...
9	Cray/SGI	T3E1200	671.2
...	...	...	...
32	Fujitsu	VPP5000/31	286.9

Et les domaines d'applications et sites d'installation correspondants:

Rang	Site	Domaine
1	Sandia National Labs, USA	Recherche
2	Lawrence Livermore National Laboratory, USA	Recherche, Energie
3	Los Alamos National Laboratory, USA	Recherche
4	Gouvernement, USA	Secret
5	Université de Tokyo, Japon	Académique
...	...	...
9	Deutscher Wetterdienst, Germany	Recherche, Météo
...	...	...
32	Météo-France, France	Recherche, Météo

Pour avoir une idée des puissances requises selon les domaines d'application, on pourra se reporter à [CS98]. En quelques mots, il faut savoir que la prévision météorologique à deux jours demande environ 100 Mégaoctets de mémoire et 100 Mégaflops, pour 3 jours il faut compter environ 1 Gigaoctet pour environ 10 Gigaflops. Le design d'une molécule pharmaceutique nécessite à l'heure actuelle environ 10 Gigaoctets pour 100 Gigaflops.

Chapitre 2

Réseau de machines sous PVM

2.1 Introduction

PVM veut dire "Parallel Virtual Machine":

- Elle contient une librairie C (et Fortran) de communication par passage de messages, cf. `http://www.epm.ornl.gov/pvm/`, développée au "Oak Ridge National Laboratory",
- Elle contient également un processus "démon" `pvmd3` (chaque type de machine doit avoir son propre pvmd3) et de bibliothèques d'interfaçage (`libpvm3.a` et `libgpvm3.a` pour le C)

Elle peut fonctionner sur les architectures suivantes:

Stations	OS	Super-calculateurs
DEC	BSD	Convex C2, Exemplar
IBM	System-5	Cray C90, T-3D
HP	Linux	Intel Paragon, Hypercube
Silicon Graphics		IBM SP-2, 3090
Sun		TMC CM-2, CM-5
NeXT		Kendall Square
Data General		Sequent
386/486/586 Unix		Maspar
		Encore
		Fujitsu
		Stardent
...	...	...

La machine PVM (voir figure 2.1) est une collection de machines réelles sur un même réseau sur lesquelles s'exécute un démon `pvmd3`. `pvmd3` est un processus UNIX qui contrôle les processus utilisateurs au sein d'une application PVM et qui coordonne les communications entre les machines qui composent la machine PVM. Un démon pvmd3 est présent par ordinateur et par utilisateur de la machine PVM et maintient la configuration globale et locale de la machine PVM.

La librairie PVM permet de communiquer avec ces démons pvmd3 pour créer et détruire des processus, gérer des tampons de communication, envoyer, recevoir (point à point ou par broadcast) des messages (de façon bloquante ou non-bloquante), synchroniser des processus (par barrières de synchronisation) connaître l'état de la configuration de la machine PVM et pour la modifier (dynamiquement)...

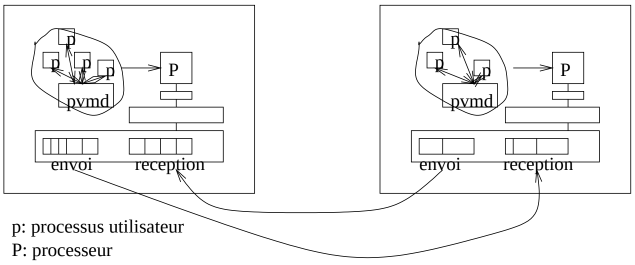


FIG. 2.1 – *La machine PVM*

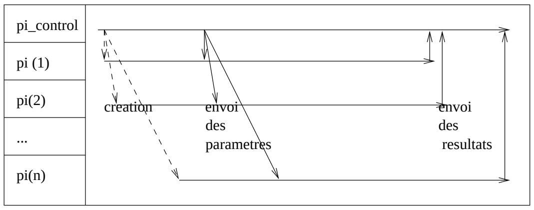


FIG. 2.2 – *Flot de contrôle du calcul parallèle de π*

2.2 Démarrer par l'exemple

Supposons que nous voulions calculer π en parallèle par la formule suivante,

$$\begin{aligned} \pi &= \int_0^1 \frac{4}{1+x^2} dx \\ &\cong \sum_{i=1}^{n} \frac{1}{n} \frac{4}{1+\left((i-\frac{1}{2})\frac{1}{n}\right)^2} \end{aligned}$$

Une solution est le paradigme *Maître/Esclave*: Un maître va lancer `N` esclaves chargés de calculer les sommes partielles,

$$P_k = \sum_{i=k*n/N}^{(k+1)*n/N} \frac{1}{n} \frac{4}{1+\left((i-\frac{1}{2})\frac{1}{n}\right)^2}$$

pour $k = 0, \cdots, \mathtt{N}-1$.

C'est une parallélisation par décomposition fonctionnelle (que l'on reverra avec la FFT, section 5.2.2). On pourrait aussi faire de la décomposition de domaine pour calculer une fonction s'appliquant indépendemment à des sous-blocs d'un tableau.

Donc le maître `pi_control` va lancer `N` copies d'un esclave `pi` avec l'entier k correspondant, puis attendre que tous aient terminé leur exécution, enfin effectuer la somme de tous les résultats partiels (voir figure 2.2).

2.2.1 La base pour démarrer: le contrôle des processus.

pvm_mytid

C'est une instruction obligatoire en début de code PVM [1] qui permet de raccrocher le processus à un démon `pvmd`. Sa syntaxe est:

```
int tid;
tid=pvm_tid();
```

Cela renvoie `tid` qui est l'identificateur de tâche du processus. Ce `tid` permet de définir de façon unique ce processus et de s'adresser à lui. Il y a un cas d'Erreur si cela renvoie `tid <0` (démon PVM inexistant par exemple).

pvm_exit

C'est une instruction (presque) obligatoire en fin de code PVM. Elle permet de dire au démon `pvmd` que la tâche quitte PVM.

```
int info;
info=pvm_exit();
```

Il y a un cas d'Erreur si cela renvoie `info <0`.

pvm_spawn

C'est l'instruction qui permet de lancer `n` tâches PVM exécutant un même code (`"programme"`):

```
int numt;
numt=pvm_spawn(char *task,char **argv,int flag,char
                              *where,int ntask,int *tids);
```

On se contentera pour le plus souvent de (voir cependant la section 5.2.2 pour une application de la forme de "`spawn`" la plus générale),

```
int numt,etid[n];
numt=pvm_spawn("programme",NULL,PvmTaskDefault,NULL,n,&etid[0]);
```

Le `tid` de chacune des tâches est rangé dans une entrée du tableau `etid`. Cette instruction Renvoie `numt <= n` qui est le nombre de tâche(s) effectivement lancée(s).

Pour les autres champs:

- `argv` est la liste d'arguments (type ligne de commande UNIX) à passer à `"programme"`,
- `flag` donne des options à `pvm_spawn`,
- pour certaines de ces options `where` indique sur quel type d'hôte lancer les processus.

pvm_parent

Cette instruction retourne `tid` valant le `tid` de son processus parent (celui qui a crée le processus courant):

```
int tid;
tid=pvm_parent();
```

Elle peut également renvoyer `tid=PvmNoParent` si le processus courant n'a pas été crée par `pvm_spawn`.

1. On pourra se reporter au livre [GBD+94] pour plus de détails.

2.2.2 La base: passage de messages

L'envoi des messages est non-bloquant. Ceci est géré grâce à un tampon d'envoi. Celui-ci doit être,

- crée (`pvm_initsend`),
- rempli (`pvm_pk...`),
- son contenu, envoyé (`pvm_send`).

La réception peut être bloquante ou non-bloquante. On a aussi un tampon pour la réception mais que l'on n'a pas besoin de créer explicitement. On peut,

- recevoir un contenu (`pvm_recv`),
- en extraire les données une à une (`pvm_upk...`),
- en tester l'état (avec `pvm_bufinfo`, surtout utile pour les réceptions non-bloquantes).

pvm_initsend

C'est une instruction obligatoire avant l'envoie d'un message: elle initialise le tampon d'envoie des données:

```
int info;
info=pvm_initsend(int encoding);
```

`encoding` est un entier permettant de spécifier comment coder les données (surtout utile en milieur hétérogène). Ici on emploiera toujours `PvmDataDefault`. Il y a une Erreur si `pvm_initsend` renvoie `info <0`.

pvm_pk...

Ce sont les fonctions qui effectuent le compactage de différentes données d'un processus, destinées à être envoyées, dans le tampon d'envoi:

```
int info;
info=pvm_pkint(int *ip,int nitem,int stride);
```

- `ip` est un tableau d'entiers d'au moins `nitem*stride` de long. `ip` peut être également un pointeur sur une variable entière.
- `nitem` est le nombre d'éléments de `ip` que l'on veut compacter dans le tampon d'envoi.
- `stride` est le décalage entre deux éléments de `ip` à compacter. Par exemple, si `int ip[6]; stride=2; nitem=3;` alors les éléments compactés de `ip` seront `ip[0], ip[2], ip[4]`.

Il y a eu une Erreur si cela renvoie `info <0`. Il y a une fonction de compactage par type de données possible, donc on trouvera également `pvm_pkfloat`, `pvm_pkdouble`, `pvm_pkstr` etc.

pvm_send

Cette instruction envoie un message de façon non-bloquante:

```
int info;
info=pvm_send(int tid,int msgtag);
```

`tid` est l'identificateur de processus du destinataire du message. `msgtag` est un entier identifiant une classe de messages. Cela permet de filtrer les messages (à la réception). Il y a un cas d'Erreur si elle renvoie `info <0`.

pvm_upk...

Elles correspondent aux instructions `pvm_pk...`:

```
int info;
info=pvm_upkint(int *ip,int nitem,int nstride);
```

Elles ont les mêmes arguments que pour les fonctions `pvm_pk...`. Il y a une Erreur si elles renvoient `info <0`.

pvm_recv

Cette instruction reçoit un message de type `msgtag` de l'expéditeur `tid`:

```
int bufid;
bufid=pvm_recv(int tid,int msgtag);
```

Elle bloque le processus receveur tant que le message n'est pas reçu. Si `tid` vaut -1, peut recevoir de n'importe qui. Si `msgtag` vaut -1, peut recevoir de n'importe qui. Il y a un cas d'Erreur si `pvm_recv` renvoie `info <0`.

2.2.3 L'exemple complet

Le code du maître est:

```
/* Maitre pi_control.c */
#include <stdio.h>
#include "pvm3.h"
#define NPROC 10
#define N 1000
#define SUMTAG 1
#define INPUTTAG 2

main()
{
int i, n, nproc, numt, bufid, tid[10];
float psum,sum;
n = N;
nproc = NPROC;
printf("I'm t%x spawning %d partial computations\n",pvm_mytid(),NPROC);
numt = pvm_spawn("pi",(char **) 0,PvmTaskDefault,"",NPROC,&tid[0]);
if (numt != NPROC)
{
  printf("could not spawn %d tasks\n",NPROC);
  exit(1);
}
 for (i=0; i<NPROC;i++)
   {
     pvm_initsend(PvmDataDefault);
     pvm_pkint(&i,1,1);
     pvm_pkint(&n,1,1);
     pvm_pkint(&nproc,1,1);
     pvm_send(tid[i],INPUTTAG);
   }
 sum = 0;
 for (i=0; i<NPROC; i++)
```

```
   {
     bufid = pvm_recv(tid[i], SUMTAG);
     pvm_upkfloat(&psum,1,1);
     printf("Partial sum received from t%x: %f\n", tid[i], psum);
     sum = sum+psum;
   }
 printf("From master process: Pi = %f\n",sum);
 pvm_exit();
 exit(0);
}
```

Le code unique des esclaves est:

```
/* Esclave pi.c */
#include "pvm3.h"
#include <stdio.h>
#define SUMTAG 1
#define INPUTTAG 2

main()
{
  int i, j, n, nproc, bufid, mtid;
  float psum;
  mtid = pvm_parent();
  bufid = pvm_recv(mtid, INPUTTAG);
  pvm_upkint(&i,1,1);
  pvm_upkint(&n,1,1);
  pvm_upkint(&nproc,1,1);
  printf("I am slave %d with parent t%x\n",i,mtid);
  psum = 0;
  for (j=(n/nproc)*i+1;j<=(n/nproc)*(i+1);j++)
    psum = psum+4.0/(1+(j-.5)*(j-.5)/n/n);
  psum = psum/n;
  pvm_initsend(PvmDataDefault);
  pvm_pkfloat(&psum,1,1);
  pvm_send(mtid, SUMTAG);
  pvm_exit();
  exit(0);
}
```

Un exemple d'exécution avec $NPROC = 10$ (nombre de processeurs, défini dans le maître) et $N = 1000$ itérations (défini également dans le maître):

```
pvm> spawn -1 -> pi_control
[1]
1 successful
t4008b
pvm> [1:t4008b] I'm t4008b spawning 10 partial computations
[1:t4008c] I am slave 0 with parent t4008b
[1:t40091] I am slave 5 with parent t4008b
[1:t40092] I am slave 6 with parent t4008b
[1:t40090] I am slave 4 with parent t4008b
[1:t40093] I am slave 7 with parent t4008b
[1:t4008b] Partial sum received from t4008c: 0.398675
[1:t4008e] I am slave 2 with parent t4008b
```

```
[1:t4008c] EOF
[1:t40092] EOF
[1:t40091] EOF
[1:t4008f] I am slave 3 with parent t4008b
[1:t40095] I am slave 9 with parent t4008b
[1:t40093] EOF
[1:t40090] EOF
[1:t4008d] I am slave 1 with parent t4008b
[1:t4008e] EOF
[1:t40095] EOF
[1:t4008f] EOF
[1:t4008b] Partial sum received from t4008d: 0.390908
[1:t4008b] Partial sum received from t4008e: 0.376245
[1:t4008b] Partial sum received from t4008f: 0.356198
[1:t4008b] Partial sum received from t40090: 0.332565
[1:t4008b] Partial sum received from t40091: 0.307088
[1:t4008b] Partial sum received from t40092: 0.281226
[1:t4008b] Partial sum received from t40093: 0.256060
[1:t4008d] EOF
[1:t40094] I am slave 8 with parent t4008b
[1:t4008b] Partial sum received from t40094: 0.232297
[1:t4008b] Partial sum received from t40095: 0.210332
[1:t4008b] From master process: Pi = 3.141593
[1:t40094] EOF
[1:t4008b] EOF
[1] finished
```

Chapitre 3

Equation de Laplace en dimension 1

Ce chapitre et le suivant présentent succinctement des méthodes de décomposition de domaine.

La résolution numérique des équations aux dérivées partielles intervient dans de nombreux domaines scientifiques et industrielles: météorologie, aéronautique, électronique, télécommunications, mathématiques financières, automobile, environnement, ...

Une des principales limitations de la précision des calculs vient de la place en mémoire vive exigée par les simulations de dispositifs complets. En effet, celle-ci ne varie pas linéairement avec la taille du domaine de calcul.

L'emploi de calculateurs parallèles permet dans une large mesure de s'affranchir de cette limite. La place mémoire disponible dépend linéairement du nombre de processeurs du calculateur. Les données de la simulation sont réparties sur les processeurs.

L'accès d'un processeur à sa zone mémoire est beaucoup plus rapide que l'accès aux mémoires des autres processeurs. Ceci oblige à repenser les algorithmes conçus pour les calculateurs "classiques" et à en proposer de nouveaux qui soient adaptés aux nouvelles architectures.

Les méthodes de décomposition de domaine présentées dans ce chapitre et le suivant en sont des exemples. On considère un problème posé sur un (grand) domaine de calcul. Ce domaine est décomposé en (petits) sous-domaines. Chaque sous-domaine est attribué à un processeur. Les problèmes sont résolus en parallèle sur chaque sous-domaine. Pour que les solutions locales ainsi calculées correspondent à la solution du problème global, il est nécessaire d'imposer un raccord continu des solutions locales. Ceci se fait à l'aide d'une méthode itérative.

Nous présentons ici quelques algorithmes de méthodes de décomposition de domaine dans le cadre de l'équation de Laplace en dimension 1 d'espace (ce chapitre) et en dimension 2 (chapitre suivant). La méthode est d'abord présentée au niveau continu puis sur le problème discrétisé. Dans chaque cas, on considère tout d'abord des conditions de raccord de Dirichlet (continuité de la solution) puis de Robin (raccord d'une combinaison linéaire du flux et de la valeur de la solution). L'intérêt de dernier choix est développé. Une décomposition en deux sous-domaines seulement est considérée.

Ces méthodes sont en fait très générales et s'appliquent à des équations plus complexes et à des géométries plus générales que celles présentées ici. Il est aussi important ce signaler que ces méthodes sont aussi très intéressantes pour des ordinateurs monoprocesseurs. Dans ce cas, le processeur traite successivement les sous-domaines. Le gain en place mémoire vient du fait que le coût d'une simulation numérique n'est pas linéaire par rapport à la taille du domaine de calcul.

3.1 Aspect mathématique

3.1.1 Solutions explicites

Nous considérons l'équation de Laplace

$$-\frac{\partial^2 u}{\partial x^2} = 0 \tag{3.1}$$

où u est une fonction scalaire de la variable d'espace $x \in [0, L]$ avec différents jeux de conditions aux limites:

$$u(0) = g_0 \text{ et } u(L) = g_L \tag{3.2}$$

ou bien

$$u(0) = g_0 \text{ et } (\frac{\partial u}{\partial x} + \alpha u)(L) = g_L, \tag{3.3}$$

α est une constante positive ou nulle et g_0, g_L sont des nombres réels. Nous désignerons par **P1** le problème (3.1)-(3.2) et par P2 le problème (3.1)-(3.3).

Etude du problème P1

L'équation (3.1) est une équation différentielle ordinaire en x dont la solution générale est

$$u(x) = a + bx, \quad a,\, b \in \mathbb{R}.$$

Les conditions aux limites (3.2) sont satisfaites si et seulement si a et b vérifient

$$\left\{ \begin{array}{r} a = g_0 \\ a + bL = g_L. \end{array} \right. \tag{3.4}$$

dont l'unique solution est

$$\left\{ \begin{array}{l} a = g_0 \\ b = \frac{g_L - g_0}{L}. \end{array} \right.$$

Le problème P1 admet donc aussi une solution unique:

$$u(x) = g_0 + \frac{g_L - g_0}{L} x = g_L + \frac{g_L - g_0}{L}(x - L). \tag{3.5}$$

Etude du problème P2

L'équation (3.1) est une équation différentielle ordinaire en x dont la solution générale est

$$u(x) = a + bx, \quad a,\, b \in \mathbb{R}.$$

Les conditions aux limites (3.3) sont satisfaites si et seulement si a et b vérifient

$$\left\{ \begin{array}{r} a = g_0 \\ b + \alpha(a + bL) = g_L. \end{array} \right. \tag{3.6}$$

dont l'unique solution est

$$\left\{ \begin{array}{l} a = g_0 \\ b = \frac{g_L - \alpha g_0}{1 + \alpha L}. \end{array} \right.$$

Le problème P2 admet donc aussi une solution unique

$$u(x) = g_0 + \frac{g_L - \alpha g_0}{1 + \alpha L}\, x.$$

Remarque 3.1.1 *L'équation de Laplace avec un second membre f dans (3.1) et les conditions aux limites (3.2) ou (3.3) se ramène au problème P1 ou P2: on introduit la fonction $w(x) = -\int_0^x (\int_0^s f(t)\, dt)\, ds$ telle que $-w_{xx} = f$. Par linéarité de l'équation de Laplace, on peut chercher la solution sous la forme $v + w$ où v satisfait l'équation (3.1) avec les second membres des conditions aux limites (3.2) ou (3.3) modifiés.*

3.2 Méthodes de décomposition de domaine - Cas continu

Nous considérons l'équation unidimensionnelle

$$-\frac{\partial^2 u}{\partial x^2} = f \tag{3.7}$$

où u et f sont des fonctions scalaires de la variable d'espace $x \in [0, L]$ avec des conditions aux limites de Dirichlet:

$$u(0) = g_0 \text{ et } u(L) = g_L \tag{3.8}$$

où g_0, $g_L \in \mathbb{R}$.

Le segment $\Omega = [0, L]$ est décomposé en deux sous-domaines $\Omega_1 = [0, L_1]$ et $\Omega_2 = [l_2, L]$, $0 < l_2 \leq L_1 < L$. Nous désignerons par **P** le problème défini par (3.7)-(3.8). Nous allons réécrire P de deux manières différentes sous la forme de deux problèmes couplés posés dans les sous-domaines Ω_1 et Ω_2. Puis, nous étudierons une méthode itérative de résolution de ces systèmes couplés adaptée aux architectures parallèles.

On désigne par u_1 la restriction de u au domaine Ω_1 et par u_2 sa restriction au domaine Ω_2. Le couple (u_1, u_2) vérifie

$$\begin{cases} -\frac{\partial^2 u_1}{\partial x^2} = f \text{ dans } \Omega_1 \\ u_1(0) = g_0 \end{cases} \tag{3.9}$$

$$\begin{cases} -\frac{\partial^2 u_2}{\partial x^2} = f \text{ dans } \Omega_2 \\ u_2(L) = g_L \end{cases} \tag{3.10}$$

D'après (3.9), la forme générale de u_1 est donnée par

$$u_1 = u_{|\Omega_1} + b_1\, x, \quad b_1 \in \mathbb{R}$$

et d'après (3.10), celle de u_2 est

$$u_2 = u_{|\Omega_2} + b_2(x - L), \quad b_2 \in \mathbb{R}.$$

On voit que les équations (3.9)-(3.10) ne suffisent pas à déterminer de manière unique le couple (u_1, u_2). Dans les paragraphes qui suivent, nous ajoutons des conditions de raccord aux interfaces (dites aussi conditions d'interface) $x = L_1$ et $x = l_2$ pour que le couple (u_1, u_2) soit déterminé de manière unique et soit égal aux restrictions de u aux sous-domaines Ω_1 et Ω_2. Pour chaque jeu de conditions d'interfaces, on propose une méthode itérative de résolution adaptée aux architectures parallèles.

3.2.1 Reformulation avec des conditions de Dirichlet - Problème P'

Nous ajoutons des conditions de raccord de type Dirichlet:

$$u_1(L_1) = u_2(L_1) \tag{3.11}$$

et

$$u_2(l_2) = u_1(l_2). \tag{3.12}$$

Nous désignerons par **P'** le problème (3.9)-(3.10)-(3.11)-(3.12). Nous allons montrer que ce problème admet un couple solution unique (u_1, u_2) si et seulement si les sous-domaines se recouvrent $(L_1 > l_2)$.

Existence

La solution u du problème P ((3.7)-(3.8)) étant continuement dérivable, le couple (u_1, u_2), où $u_1 = u_{|\Omega_1}$ et $u_2 = u_{|\Omega_2}$, est solution du problème P'. Remarquons que s'il y a unicité, la solution de P' est la restriction de u aux sous-domaines.

Unicité

Par linéarité des équations, il suffit de considérer les équations homogènes $g_0 = g_L = f = 0$. D'après (3.9) et (3.10), on a

$$u_1(x) = b_1 x$$

et

$$u_2(x) = b_2(x - L).$$

où b_1, $b_2 \in \mathbb{R}$. Les conditions de raccord (3.11)-(3.12) deviennent:

$$b_1 L_1 = b_2(L_1 - L)$$

et

$$b_2(l_2 - L) = b_1 l_2.$$

Il s'agit d'un système en b_1, b_2:

$$\begin{cases} L_1 b_1 - (L_1 - L) b_2 = 0 \\ -b_1 l_2 + b_2(l_2 - L) = 0. \end{cases} \tag{3.13}$$

Le déterminant de (3.13) vaut $L(l_2 - L_1)$.

Il est non nul ssi les sous-domaines se recouvrent, $l_2 < L_1$, (rappelons que l'on a déjà $l_2 \leq L_1$). Alors, on a $b_1 = b_2 = 0$, $u_1(x) = 0$ et $u_2(x) = 0$. Ceci prouve l'unicité. Dans le cas où les sous-domaines sont adjacents ($l_2 = L_1$), le déterminant de (3.13) s'annule, et le système linéaire (3.13) admet une infinité de solutions $b_1 = \frac{l_2 - L}{l_2} b_2$, $b_2 = b_2$. Il en est alors de même du problème P' qui admet une infinité à un paramètre de solutions. Ces solutions sont affines dans chacun des sous-domaines et se raccordent en $L_1 = l_2$ (voir Figure 3.1).

On a ainsi montré le

Lemme 3.2.1 *Le problème P' (Eqs. (3.9)-(3.10)-(3.11)-(3.12)) admet une unique solution ssi les sous-domaines se recouvrent. Dans ce cas, la solution (u_1, u_2) de P' est composée des restrictions, aux sous-domaines, de la solution u du problème P (i.e. $u_1 = u$ sur Ω_1 et $u_2 = u$ sur Ω_2).*

3.2.2 Algorithme parallèle de résolution du problème P' (Algorithme de Schwarz)

Nous considérons une méthode itérative de résolution du problème P'. On part de u_1^0 et u_2^0 quelconques et on pose

$$\begin{cases} -\frac{\partial^2 u_1^{n+1}}{\partial x^2} = f \text{ dans } \Omega_1 \\ u_1^{n+1}(0) = g_0 \\ u_1^{n+1}(L_1) = u_2^n(L_1) \end{cases} \tag{3.14}$$

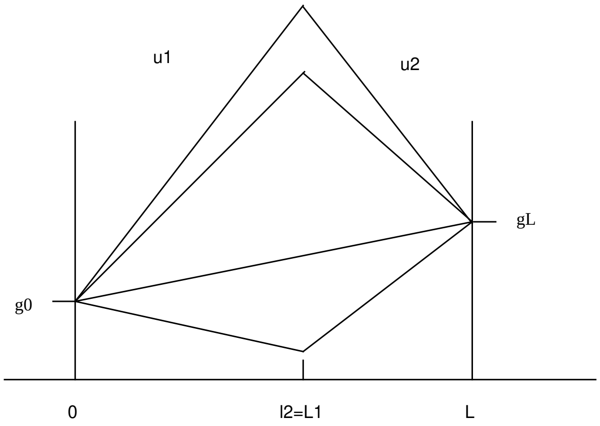


FIG. 3.1 – *Solutions multiples dans le cas sans recouvrement $l_2 = L_1$.*

$$\left\{\begin{array}{l} -\frac{\partial^2 u_2^{n+1}}{\partial x^2} = f \text{ dans } \Omega_2 \\ u_2^{n+1}(L) = g_L \\ u_2^{n+1}(l_2) = u_1^n(l_2) \end{array}\right. \tag{3.15}$$

Les problèmes définissant l'algorithme sont du type P1 et sont donc bien posés.

Remarque 3.2.1 *L'algorithme de Schwarz est bien adapté au calcul parallèle. On peut affecter un processeur à chaque sous-domaine. Les résolutions des problèmes de Laplace dans les sous-domaines sont indépendantes et peuvent être effectuées en parallèle. Une fois ce calcul terminé, chaque sous-domaine envoie à son voisin la valeur dont le voisin a besoin pour mettre à jour le second membre de son problème de Dirichlet (i.e. les valeurs courantes de $u_1(l_2)$ et $u_2(L_1)$). La quantité de données à transmettre est très petite par rapport à la quantité de données à traiter par chaque sous-domaine. Il s'agit d'un algorithme à gros grain. La traduction en PVM de cet algorithme est très simple. On a tout d'abord un programme maître qui est chargé de lancer un esclave par sous-domaine:*

```
#include <stdio.h>
#include "pvm3.h"

/* Tag des messages d'initialisation */
#define INIT 0

/* Nombre de sous-domaines */
#define NB_DOMAINES 2

/* Condition aux limites gauche */
double g0;

/* Condition aux limites droite */
double gL;

/* tids des esclaves */
int esclaves[NB_DOMAINES];

int n;
double d;
```

```
int main()
{
/* par exemple... */
   g0=1;
   gL=2;

/* on se raccroche au demon pvmd */
   if (pvm_mytid() < 0)
    {
      printf("Impossible de lancer le maitre\n");
      exit(1);
    }

/* on cree les processus esclaves */
   n=pvm_spawn("esclave",NULL,PvmTaskDefault,NULL,NB_DOMAINES,
                                                  &esclaves[0]);
   if (n < NB_DOMAINES)
    {
      printf("Impossible de lancer tous les esclaves\n");
      pvm_exit();
      exit(1);
    }

/* on envoie les noms des voisins et les
   conditions aux limites par sous-domaine */

/* d'abord le bord gauche du domaine total */
   pvm_initsend(PvmDataDefault);
   n=-1;
   pvm_pkint(&n,1,1);
   pvm_pkint(&esclaves[1],1,1);
   pvm_pkdouble(&g0,1,1);
   d=0;
   pvm_pkdouble(&d,1,1);
   pvm_send(esclaves[0],INIT);

/* ensuite l'interieur du domaine total */
   for (i=1;i<NB_DOMAINES-1;i++)
    {
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&esclaves[i-1],1,1);
      pvm_pkint(&esclaves[i+1],1,1);
      d=0;
      pvm_pkdouble(&d,1,1);
      pvm_pkdouble(&d,1,1);
      pvm_send(esclaves[i],INIT);
    }

/* enfin le bord droit du domaine total */
   pvm_initsend(PvmDataDefault);
   pvm_pkint(&esclaves[NB_DOMAINES-2],1,1);
   n=-1;
   pvm_pkint(&n,1,1);
```

```
   d=0;
   pvm_pkdouble(&d,1,1);
   pvm_pkdouble(&gL,1,1);
   pvm_send(esclaves[NB_DOMAINES-1],1,1);

/* on termine */
   pvm_exit();
   exit(0);
}
```

Chaque esclave "esclave.c" a pour structure:

```
#include <stdio.h>
#include "pvm3.h"

/* Tag des messages d'initialisation */
#define INIT 0

/* Tag des messages inter-esclaves */
#define INTER 1

/* Nombre d'iterations pour Schwarz */
#define ITER 1000

/* echantillonage de la solution u
   (voir discretisation )*/
#define N 100

/* tid des voisins, egal a -1 si n'existe pas */
int voisin_gauche, voisin_droit;

/* solution u discretisee a une etape donnee */
double u[N];
int i;

void resolution()
/* resoud le probleme dans le sous-domaine */
/* a partir des conditions aux limites u[0], u[N-1] */
{
   ...
}

int main()
{
/* reception des parametres */
   pvm_recv(-1,INIT);
   pvm_upkint(&voisin_gauche,1,1);
   pvm_upkint(&voisin_droit,1,1);
   pvm_upkdouble(&u[0],1,1);
   pvm_upkdouble(&u[N-1],1,1);

/* Algorithme de Schwarz */
   for (i=0;i<ITER;i++)
    {
```

```
    /* calcul de la solution dans le sous-domaine */
      resolution();

    if (voisin_gauche>0)
    /* echanges avec le voisin gauche */
     {
       pvm_initsend(PvmDataDefault);
       pvm_pkdouble(&u[0],1,1);
       pvm_send(voisin_gauche,INTER);
     }

    if (voisin_droit>0)
    /* echange avec le voisin droit */
     {
       pvm_pkdouble(&u[N-1],1,1);
       pvm_send(voisin_droit,INTER);
     }

    if (voisin_gauche>0)
    /* reception du voisin gauche */
     {
       pvm_recv(voisin_gauche,INTER);
       pvm_upkdouble(&u[0],1,1);
     }

    if (voisin_droit>0)
    /* reception du voisin droit */
     {
       pvm_recv(voisin_droit,INTER);
       pvm_upkdouble(&u[N],1,1);
     }
    }

/* affichage des resultats */
...

/* fin */
   pvm_exit();
   exit(0);
}
```

Remarques: On aurait pu passer les paramètres aux esclaves directement à leur création. Dans la pratique également, le maître serait chargé d'attendre la fin de chaque esclave et de réceptionner leurs calculs pour affichage ou dépouillement.

Il faut aussi remarquer que `pvm_send` *étant non-bloquant alors que* `pvm_recv` *est bloquant, il vaut mieux que chaque esclave envoie ses valeurs limites, avant d'attendre les informations de ses voisins (le contraire produirait un interblocage).*

Dans la pratique enfin, le nombre d'itérations ne serait sans doute pas fixé; le test d'arrêt serait plutôt exprimé par une condition de convergence.

Remarque 3.2.2 *Une solution stationnaire de l'algorithme est solution du problème P'. Ceci prouve que si l'algorithme converge, il converge vers l'unique solution (quand les sous-domaines se chevauchent) de P' qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution u du problème P. L'algorithme de Schwarz (3.14)-(3.15) sera donc une méthode itérative de résolution du problème P.*

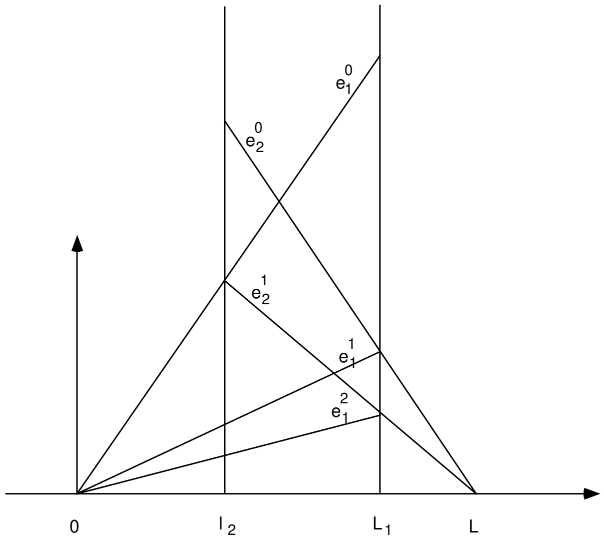


FIG. 3.2 – *Evolution de l'erreur dans l'algorithme de Schwarz.*

Pour étudier la convergence, on pose $e_1^n = u_1^n - u_1$ et $e_2^n = u_2^n - u_2$ où (u_1, u_2) est la solution du problème P'. Par linéarité des équations, l'erreur (e_1^n, e_2^n) vérifie

$$\left\{\begin{array}{l} -\frac{\partial^2 e_1^{n+1}}{\partial x^2} = 0 \text{ dans } \Omega_1 \\ e_1^{n+1}(0) = 0 \\ e_1^{n+1}(L_1) = e_2^n(L_1) \end{array}\right. \tag{3.16}$$

$$\left\{\begin{array}{l} -\frac{\partial^2 e_2^{n+1}}{\partial x^2} = 0 \text{ dans } \Omega_2 \\ e_2^{n+1}(L) = 0 \\ e_2^{n+1}(l_2) = e_1^n(l_2) \end{array}\right. \tag{3.17}$$

On a ainsi, pour $n \geq 2$, $e_1^n = \gamma_1^n x$ et $e_2^n = \gamma_2^n(x - L)$. Les conditions de Dirichlet en $x = L_1$ et $x = l_2$ donnent

$$\begin{array}{r} \gamma_1^{n+1} L_1 = \gamma_2^n (L_1 - L) \\ \gamma_2^{n+1}(l_2 - L) = \gamma_1^n \, l_2. \end{array}$$

On a donc

$$\gamma_1^{n+1} = \frac{L_1 - L}{L_1}\gamma_2^n = \frac{L_1 - L}{L_1}\,\frac{l_2}{l_2 - L}\gamma_1^{n-1} = \frac{l_2}{L_1}\,\frac{L - L_1}{L - l_2}\gamma_1^{n-1} \equiv \beta\gamma_1^{n-1}.$$

La relation est aussi valable pour γ_2^{n+1}. Comme $l_2 \leq L_1$, le terme β est toujours positif et on a: $\beta < 1 \iff l_2 < L_1$. Pour $l_2 = L_1$, on a $\beta = 1$ et l'algorithme stagne. On a prouvé le

Théorème 3.2.2 *L'algorithme de Schwarz (3.14)-(3.15) converge si et seulement si les sous-domaines se recouvrent ($l_2 < L_1$).*

Remarque 3.2.3 *L'absence de convergence dans le cas sans recouvrement est en accord avec le fait que le problème P' est alors mal posé.*

Remarque 3.2.4 *Pour un petit recouvrement, $L_1 = l_2 + \epsilon$, $\epsilon \ll 1$, β dépend des pourcentages de recouvrement*

$$\beta \simeq 1 - \left(\frac{\epsilon}{L_1} + \frac{\epsilon}{L - l_2}\right).$$

Plus le recouvrement est faible, plus la convergence est lente.

Remarque 3.2.5 *La figure 3.2 donne une interprétation géométrique de la convergence.*

3.2.3 Reformulation avec des conditions de Robin - Problème P"

Pour fermer le système d'équations (3.9)-(3.10), nous ajoutons des conditions de raccord de type Robin:

$$(\frac{\partial u_1}{\partial n_1} + \alpha u_1)(L_1) = (-\frac{\partial u_2}{\partial n_2} + \alpha u_2)(L_1) \tag{3.18}$$

et

$$(\frac{\partial u_2}{\partial n_2} + \alpha u_2)(l_2) = (-\frac{\partial u_1}{\partial n_1} + \alpha u_1)(l_2) \tag{3.19}$$

où α est une constante strictement positive, n_1 (resp. n_2) désigne la normale extérieure au domaine Ω_1 (resp. Ω_2). Les signes $-$ devant les dérivées normales viennent du fait que $n_2 = -n_1$. Nous désignerons par **P"** le problème (3.9)-(3.18)- (3.10)-(3.19). Nous allons montrer que ce problème admet un couple solution unique (u_1, u_2) avec ou sans recouvrement des sous-domaines.

Existence

La solution u du problème P étant continuement dérivable, le couple (u_1, u_2), où $u_1 = u_{|\Omega_1}$ et $u_2 = u_{|\Omega_2}$, est solution du problème P". Remarquons que s'il y unicité la solution de P" est la restriction de u aux sous-domaines.

Unicité

Par linéarité des équations, il suffit de considérer le cas homogène $f = g_0 = g_L = 0$. D'après (3.9) et (3.10), on a

$$u_1(x) = b_1 x$$

et

$$u_2(x) = b_2(x - L).$$

où b_1, $b_2 \in \mathbb{R}$. Les conditions de raccord (3.18)-(3.19) deviennent:

$$b_1 + \alpha b_1 L_1 = b_2 + \alpha b_2(L_1 - L)$$

et

$$-b_2 + \alpha b_2(l_2 - L) = -b_1 + \alpha b_1 l_2$$

Il s'agit d'un système en b_1, b_2:

$$\begin{cases} (1 + \alpha L_1)b_1 - (1 - \alpha(L - L_1))b_2 = 0 \\ (-1 + \alpha l_2)b_1 + (1 + \alpha(L - l_2))b_2 = 0. \end{cases} \tag{3.20}$$

Le déterminant de (3.20) vaut $\alpha L(2 + \alpha(L_1 - l_2))$ et est strictement positif. Le système (3.20) admet comme unique solution la solution nulle. Il en est alors de même du problème P". On a ainsi montré le

Lemme 3.2.3 *Le problème P" (Eqs. (3.9)-(3.18)-(3.10)-(3.19)) admet une unique solution. La solution (u_1, u_2) de P" est composée des restrictions, aux sous-domaines, de la solution u du problème P (i.e. $u_1 = u$ sur Ω_1 et $u_2 = u$ sur Ω_2).*

Remarque 3.2.6 *Il est important de noter que le problème P", contrairement au problème P', admet une solution unique même quand les sous-domaines ne se recouvrent pas.*

3.2.4 Algorithme parallèle de résolution du problème P"

Nous considérons une méthode itérative de résolution du problème P". On part de u_1^0 et u_2^0 quelconques et on pose

$$\left\{\begin{array}{l} -\dfrac{\partial^2 u_1^{n+1}}{\partial x^2} = f \text{ dans } \Omega_1 \\ u_1^{n+1}(0) = g_0 \\ (\dfrac{\partial u_1^{n+1}}{\partial n_1} + \alpha u_1^{n+1})(L_1) = (-\dfrac{\partial u_2^n}{\partial n_2} + \alpha u_2^n)(L_1) \end{array}\right. \tag{3.21}$$

$$\left\{\begin{array}{l} -\dfrac{\partial^2 u_2^{n+1}}{\partial x^2} = f \text{ dans } \Omega_2 \\ u_2^{n+1}(L) = g_L \\ (\dfrac{\partial u_2^{n+1}}{\partial n_2} + \alpha u_2^{n+1})(l_2) = (-\dfrac{\partial u_1^n}{\partial n_1} + \alpha u_1^n)(l_2) \end{array}\right. \tag{3.22}$$

Les problèmes aux limites définissant l'algorithme sont du type P2 et donc bien posés.

Remarque 3.2.7 *Une solution stationnaire de l'algorithme est solution du problème P". Ceci prouve que si l'algorithme converge (voir plus loin), il converge vers l'unique solution de P" qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution u du problème P. L'algorithme (3.21)-(3.22) sera donc une méthode itérative de résolution du problème P.*

Pour étudier la convergence, on pose $e_1^n = u_1^n - u_1$ et $e_2^n = u_2^n - u_2$ où (u_1, u_2) est la solution du problème P". L'erreur (e_1^n, e_2^n) vérifie par linéarité des équations

$$\left\{\begin{array}{l} -\dfrac{\partial^2 e_1^{n+1}}{\partial x^2} = 0 \text{ dans } \Omega_1 \\ e_1^{n+1}(0) = 0 \\ (\dfrac{\partial e_1^{n+1}}{\partial n_1} + \alpha e_1^{n+1})(L_1) = (-\dfrac{\partial e_2^n}{\partial n_2} + \alpha e_2^n)(L_1) \end{array}\right. \tag{3.23}$$

$$\left\{\begin{array}{l} -\dfrac{\partial^2 e_2^{n+1}}{\partial x^2} = 0 \text{ dans } \Omega_2 \\ e_2^{n+1}(L) = 0 \\ (\dfrac{\partial e_2^{n+1}}{\partial n_2} + \alpha e_2^{n+1})(l_2) = (-\dfrac{\partial e_1^n}{\partial n_1} + \alpha e_1^n)(l_2) \end{array}\right. \tag{3.24}$$

On a ainsi, pour $n \geq 2$, $e_1^n = \gamma_1^n x$ et $e_2^n = \gamma_2^n (x - L)$. Les conditions de Robin en $x = L_1$ et $x = l_2$ donnent

$$\begin{array}{c} \gamma_1^{n+1} + \alpha\gamma_1^{n+1} L_1 = \gamma_2^n + \alpha\gamma_2^n (L_1 - L) \\ -\gamma_2^{n+1} + \alpha\gamma_2^{n+1}(l_2 - L) = -\gamma_1^n + \alpha\gamma_1^n\, l_2. \end{array}$$

On a donc

$$\gamma_1^{n+1} = \frac{1 + \alpha(L_1 - L)}{1 + \alpha L_1}\,\gamma_2^n = \frac{1 + \alpha(L_1 - L)}{1 + \alpha L_1}\;\frac{-1 + \alpha l_2}{-1 + \alpha(l_2 - L)}\,\gamma_1^{n-1} = \beta\gamma_1^{n-1}.$$

où

$$\beta = \frac{1 - \alpha l_2}{1 + \alpha L_1}\;\frac{1 - \alpha(L - L_1)}{1 + \alpha(L - l_2)}.$$

La relation est aussi valable pour γ_2^{n+1}. Le paramètre α étant positif, on peut montrer qu'avec ou sans recouvrement des sous-domaines,

$$-1 < \beta < 1.$$

On a ainsi prouvé le

Théorème 3.2.4 *Pour $\alpha > 0$, l'algorithme (3.21)-(3.22) converge.*

Remarque 3.2.8 *Contrairement à l'algorithme de Schwarz, le recouvrement des sous-domaines n'est pas nécessaire pour assurer la convergence de l'algorithme. Dans le cas sans recouvrement ($l_2 = L_1$), on a*

$$\beta = \frac{1-\alpha l_2}{1+\alpha l_2}\,\frac{1-\alpha(L-l_2)}{1+\alpha(L-l_2)}.$$

Remarque 3.2.9 *Il est possible d'annuler β et donc d'avoir une convergence en deux itérations en choisissant $\alpha = 1/l_2$ ou bien $\alpha = 1/(L-L_1)$.*

3.3 Aspect numérique

3.3.1 Approximations par différences finies

Nous considérons l'équation

$$-\frac{\partial^2 u}{\partial x^2} = f, \quad x \in [0, L] \tag{3.25}$$

avec les conditions aux bords

$$u(0) = g_0 \text{ et } (\frac{\partial u}{\partial x} + \alpha u)(L) = g_L, \tag{3.26}$$

où α est une constante positive ou nulle et g_0, g_L sont des nombres réels. Nous introduisons des schémas d'approximation par différence finie. Nous cherchons à calculer une approximation de la solution aux points $(x_j)_{1\leq j\leq N}$ définis par

$$x_j = (j-1)\Delta x$$

où N est le nombre de points de discrétisation et $\Delta x = L/(N-1)$ désigne le pas de discrétisation. On note u_j l'approximation numérique de u au point x_j. Pour approcher (3.25), on utilise un schéma centré

$$-\frac{u_{j+1} - 2u_j + u_{j-1}}{(\Delta x)^2} = f(x_j). \tag{3.27}$$

La condition de Dirichlet dans (3.26) est approchée par

$$u_1 = g_0. \tag{3.28}$$

La condition de Robin dans (3.26) peut être approchée de plusieurs façons. Nous en mentionnons deux.

Possibilité n^{0}1.

$$\frac{u_N - u_{N-1}}{\Delta x} + \alpha u_N = g_L. \tag{3.29}$$

Cette formule correspond à une approximation de la dérivée première en x à l'aide d'un développement de Taylor à l'ordre 1:

$$u(x_{N-1}) = u(x_N) - \Delta x \frac{\partial u}{\partial x}(x_N) + O(\Delta x^2). \tag{3.30}$$

Possibilité n^{0}2.

$$\frac{u_N - u_{N-1}}{\Delta x} - \frac{\Delta x}{2} f(x_N) + \alpha u_N = g_L. \tag{3.31}$$

Cette formule correspond à une approximation de la dérivée première en x à l'aide d'un développement de Taylor à l'ordre 2:

$$u(x_{N-1}) = u(x_N) - \Delta x \frac{\partial u}{\partial x}(x_N) + \frac{1}{2}(\Delta x)^2 \frac{\partial^2 u}{\partial x^2}(x_N) + O(\Delta x^3). \tag{3.32}$$

dans lequel on a approché $\dfrac{\partial^2 u}{\partial x^2}(x_N)$ à l'aide de l'équation (3.27) en $j = N$.

La possibilité (3.31) est plus précise que la possibilité (3.29) puisqu'elle est en $O(\Delta x^2)$ alors que (3.29) est en $O(\Delta x)$.

Attention! pour estimer l'ordre on réécrit (3.30) et (3.32) sous la forme:

$$\frac{u(x_N) - u(x_{N-1})}{\Delta x} = \frac{\partial u}{\partial x}(x_N) + O(\Delta x)$$

et

$$\frac{u(x_N) - u(x_{N-1})}{\Delta x} = \frac{\partial u}{\partial x}(x_N) - \frac{1}{2}\Delta x \frac{\partial^2 u}{\partial x^2}(x_N) + O(\Delta x^2)$$

respectivement.

Remarque 3.3.1 *L'inversibilité des systèmes linéaires résultants de (3.27)-(3.28)-(3.29) ou bien de (3.27)-(3.28)-(3.31) et la convergence de u_j vers $u(x_j)$ quand Δx tend vers zéro sont prouvées dans le polycopié: [LL].*

3.3.2 Systèmes matriciels de discrétisation

Nous considérons les équations aux différences finies (3.27), (3.28), (3.29) ou (3.31). Nous introduisons des systèmes linéaires correspondants à ces schémas. Plusieurs choix sont possibles pour la matrice représentant les équations aux differences finies.

Choix n^{0}1.

On introduit le vecteur des inconnues $U = (u_j)_{1 \le j \le N}$. On écrit sans modification les équations aux différences finies (3.27), (3.28). Si on discrétise la condition de Robin par (3.29), on a

$$\begin{pmatrix}
1 & & & & & \\
-\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & & \\
& \ddots & \ddots & \ddots & & \\
& & \ddots & \ddots & \ddots & \\
& & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} \\
& & & & -\frac{1}{\Delta x} & \frac{1}{\Delta x} + \alpha
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}$$

$$
= \begin{pmatrix} g_0 \\ f(x_2) \\ \vdots \\ \vdots \\ f(x_{N-1}) \\ g_L \end{pmatrix} \tag{3.33}
$$

Si on discrétise la condition de Robin par (3.31), la matrice de discrétisation est inchangée mais le second membre est différent:

$$
\begin{pmatrix}
1 & & & & & \\
-\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & & \\
 & \ddots & \ddots & \ddots & & \\
 & & \ddots & \ddots & \ddots & \\
 & & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} \\
 & & & & -\frac{1}{\Delta x} & \frac{1}{\Delta x} + \alpha
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}
= \begin{pmatrix} g_0 \\ f(x_2) \\ \vdots \\ \vdots \\ f(x_{N-1}) \\ g_L + \frac{\Delta x}{2} f(x_N) \end{pmatrix} \tag{3.34}
$$

Il est important de remarquer que les premières et dernières lignes des matrices de (3.33) et (3.34) les rendent non symétriques.

Choix n^{0}2.

On modifie la deuxième ligne des systèmes linéaires du choix précédent en substituant u_1 par g_0 dans la deuxième ligne qui s'écrit alors:

$$
\frac{2\,u_2}{(\Delta x)^2} - \frac{u_3}{(\Delta x)^2} = f(x_2) + \frac{g_0}{(\Delta x)^2}
$$

La dernière ligne de (3.33) ou de (3.34) est divisée par Δx. Finalement, si la condition de Robin (3.26) est discrétisée par (3.29) on a le système matriciel

$$
\begin{pmatrix}
1 & & & & & \\
 & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & & \\
 & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & \\
 & & \ddots & \ddots & \ddots & \\
 & & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} \\
 & & & & -\frac{1}{(\Delta x)^2} & \frac{1}{(\Delta x)^2} + \frac{\alpha}{(\Delta x)^2}
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}
$$

$$= \begin{pmatrix} g_0 \\ f(x_2) + \frac{g_0}{(\Delta x)^2} \\ f(x_3) \\ \vdots \\ f(x_{N-1}) \\ \frac{g_L}{\Delta x} \end{pmatrix}. \quad (3.35)$$

Si on discrétise la condition de Robin par (3.31), la matrice de discrétisation est inchangée mais le second membre est différent:

$$\begin{pmatrix} 1 & & & & & \\ & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & & \\ & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} \\ & & & & -\frac{1}{(\Delta x)^2} & \frac{1}{(\Delta x)^2} + \frac{\alpha}{(\Delta x)^2} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}$$

$$= \begin{pmatrix} g_0 \\ f(x_2) + \frac{g_0}{(\Delta x)^2} \\ f(x_3) \\ \vdots \\ f(x_{N-1}) \\ \frac{g_L}{\Delta x} + \frac{f(x_N)}{2} \end{pmatrix}. \quad (3.36)$$

Il est important de remarquer la supériorité du choix $n^0 2$, qui conduit aux systèmes linéaires symétriques (3.35) et (3.36), par rapport au choix $n^0 1$ qui conduit aux systèmes non symétriques (3.33) et (3.34).

3.4 Méthodes de décomposition de domaine - Cas discret

Dans ce paragraphe, nous donnons un équivalent discret des méthodes de décomposition de domaine du cas continu traité dans le paragraphe 3.2.
Nous considérons le système linéaire unidimensionnel issu de la discrétisation du problème P (Eqs (3.7)-(3.8)):

$$\begin{pmatrix} 1 & & & & & \\ 0 & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & & \\ & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & 0 \\ & & & & & 1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix}$$

$$= \begin{pmatrix} g_0 \\ f(x_2) + \frac{g_0}{(\Delta x)^2} \\ f(x_3) \\ \vdots \\ f(x_{N-1}) + \frac{g_L}{(\Delta x)^2} \\ g_L \end{pmatrix}. \quad (3.37)$$

où $(u_j)_{1 \leq j \leq N}$ est un vecteur de réels qui sera noté U. Rappelons que les conditions de Dirichlet (3.8) sont implicitement contenues dans les premières et dernières lignes

de (3.37). Nous désignerons par $\mathbf{P}_h$ le problème (3.37).
L'ensemble des indices $\mathcal{N} = \{1, 2, \ldots, N\}$ est décomposé en deux sous-ensembles d'indices $\mathcal{N}_1 = \{1, 2, \ldots, N_1\}$ et $\mathcal{N}_2 = \{N_2, N_2+1, \ldots, N\}$ où N_1 et N_2 sont des entiers satisfaisants $N_2 \leq N_1$. Cela correspond à une décomposition du segment $[0, L]$ en deux sous-domaines $[0, L_1]$ et $[l_2, L]$ où $L_1 = (N_1 - 1)\Delta x$ et $l_2 = (N_2 - 1)\Delta x$. Si $N_2 < N_1$, on dira, par analogie avec le cas continu, que les sous-domaines se recouvrent. Nous allons réécrire P_h de deux manières différentes sous la forme de deux problèmes couplés posés sur les sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$. Nous considérons tout d'abord le couplage par des conditions de Dirichlet puis par des conditions de Robin discrétisées. Puis, nous étudierons une méthode itérative de résolution de ces systèmes couplés adaptée aux architectures parallèles. Pour cela, nous suivrons un plan parallèle celui du § 3.2 où le cas continu est traité.

On désigne par U_1 la restriction de U au sous-ensemble d'indices $\mathcal{N}_1$ et par U_2 sa restriction à $\mathcal{N}_2$. Le vecteur $U_1 = (u_{1,j})_{j \in \mathcal{N}_1}$ vérifie

$$A_1 \, U_1 = \begin{pmatrix} g_0 \\ f(x_2) + \frac{g_0}{(\Delta x)^2} \\ f(x_3) \\ \vdots \\ f(x_{N_1 - 1}) \end{pmatrix} \tag{3.38}$$

où

$$A_1 = \begin{pmatrix} 1 & & & & \\ & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & \\ & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & \\ & & \ddots & \ddots & \ddots \\ & & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} \end{pmatrix} \tag{3.39}$$

est une matrice rectangulaire $(N_1 - 1) \times N_1$. Le vecteur $U_2 = (u_{2,j})_{j \in \mathcal{N}_2}$ vérifie

$$A_2 \, U_2 = \begin{pmatrix} f(x_{N_2+1}) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) + \frac{g_L}{(\Delta x)^2} \\ g_L \end{pmatrix}. \tag{3.40}$$

où

$$A_2 = \begin{pmatrix} -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & -\frac{1}{(\Delta x)^2} & & \\ & \ddots & \ddots & \ddots & \\ & & -\frac{1}{(\Delta x)^2} & \frac{2}{(\Delta x)^2} & \\ & & & & 1 \end{pmatrix} \tag{3.41}$$

est une matrice rectangulaire $(N_2 - 1) \times N_2$.
Les systèmes linéaires (3.38)-(3.40) sont donc sous-déterminés. Dans les paragraphes qui suivent, nous ajoutons des conditions de raccord aux interfaces en $j = N_1$ et $j = N_2$ pour que le couple (U_1, U_2) soit déterminé de manière unique et soit égal aux restrictions de U aux sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$. Pour chaque jeu de conditions d'interfaces, on propose une méthode itérative de résolution adaptée aux architectures parallèles.

3.4.1 Reformulation avec des conditions de Dirichlet - Problème $\mathbf{P'}_h$

Nous ajoutons les conditions de raccord de type Dirichlet:

$$u_{1,N_1} = u_{2,N_1} \tag{3.42}$$

et

$$u_{2,N_2} = u_{1,N_2}. \tag{3.43}$$

Nous désignerons par **P'**$_h$ le problème (3.38)-(3.40)-(3.42)-(3.43) Nous allons montrer que ce problème admet un couple solution unique (U_1, U_2) si et seulement si les sous-domaines se recouvrent $(N_1 > N_2)$.

Existence

On construit une solution de P'$_h$ à partir de la solution U du système (3.37) en posant $U_1 = (u_j)_{j\in\mathcal{N}_1}$ et $U_2 = (u_j)_{j\in\mathcal{N}_2}$.

Remarquons que s'il y a unicité, la solution de P'$_h$ est la restriction de U aux sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$.

Unicité

Par linéarité des équations, il suffit de considérer le cas homogène $g_0 = g_L = f = 0$. Alors, d'après (3.38) et (3.40), on a

$$U_1 = (b_1(j-1)\Delta x)_{j\in\mathcal{N}_1}$$

et

$$U_2 = (b_2(j-N)\Delta x)_{j\in\mathcal{N}_2}$$

où b_1, $b_2 \in \mathbb{R}$. Les conditions de raccord (3.42)-(3.43) deviennent:

$$b_1(N_1-1)\Delta x = b_2(N_1-N)\Delta x$$

et

$$b_2(N_2-N)\Delta x = b_1(N_2-1)\Delta x.$$

Il s'agit d'un système en b_1, b_2:

$$\left\{\begin{array}{l} b_1(N_1-1)\Delta x - b_2(N_1-N)\Delta x = 0 \\ -b_1(N_2-1)\Delta x + b_2(N_2-N)\Delta x = 0. \end{array}\right. \tag{3.44}$$

Le déterminant de (3.44) vaut $(N-1)\Delta x\,(N_2-N_1)\Delta x{=}L(l_2-L_1)$.

Il est non nul si et seulement si les sous-domaines se recouvrent, $N_2 < N_1$. Alors, on a $b_1 = b_2 = 0$, $U_1 = 0$ et $U_2 = 0$. La solution de P'$_h$ est alors unique.

Dans le cas où $N_1 = N_2$, le déterminant de (3.44) s'annule. Le système linéaire admet une infinité de solutions $b_1 = \frac{(N_1-N)}{(N_1-1)}b_2, b_2 = b_2$. Il en est alors de même du problème P'$_h$ qui admet une infinité à un paramètre de solutions. Ces solutions sont affines dans chacun des sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$.

On a ainsi montré le

Lemme 3.4.1 *Le problème P'$_h$ (Eqs. (3.38)-(3.40)-(3.42)-(3.43)) admet une unique solution ssi les sous-domaines se recouvrent $N_2 < N_1$. Dans ce cas, la solution (U_1, U_2) du problème P'$_h$ est composée des restrictions aux sous-ensembles d'indices de la solution U du problème P_h (i.e. $U_1 = U_{|\mathcal{N}_1}$ et $U_2 = U_{|\mathcal{N}_2}$).*

3.4.2 Algorithme parallèle de résolution du problème P'$_h$ (Algorithme de Schwarz discret)

Nous considérons une méthode itérative de résolution du problème P'$_h$. On part de U_1^0 et U_2^0 quelconques et on pose

$$\left\{\begin{array}{l} A_1\,U_1^{n+1} = \begin{pmatrix} g_0 \\ f(x_2) + \frac{g_0}{(\Delta x)^2} \\ f(x_3) \\ \vdots \\ f(x_{N_1-1}) \end{pmatrix}, \\ u_{1,N_1}^{n+1} = u_{2,N_1}^{n} \end{array}\right. \tag{3.45}$$

et

$$\left\{\begin{array}{l} A_2\,U_2^{n+1} = \begin{pmatrix} f(x_{N_2+1}) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) + \frac{g_L}{(\Delta x)^2} \\ g_L \end{pmatrix}, \\ u_{2,N_2}^{n+1} = u_{1,N_2}^n . \end{array}\right. \tag{3.46}$$

Remarque 3.4.1 *Une solution stationnaire de l'algorithme est solution du problème P'_h. Ceci prouve que si l'algorithme converge (voir plus loin la discussion suivant le chevauchement des sous-domaines), il converge vers l'unique solution de P'_h qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution u du problème P_h. L'algorithme de Schwarz (3.45)-(3.46) sera donc une méthode itérative de résolution du problème P_h.*

Pour étudier la convergence, on pose $E_1^n = (e_{1,j}^n)_{j\in\mathcal{N}_1} = U_1^n - U_1$ et $E_2^n = (e_{2,j}^n)_{j\in\mathcal{N}_2} = U_2^n - U_2$ où (U_1, U_2) est la solution du problème P'$_h$. Par linéarité des équations, l'erreur vérifie

$$\left\{\begin{array}{l} A_1\,E_1^{n+1} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \\ e_{1,N_1}^{n+1} = e_{2,N_1}^n \end{array}\right. \tag{3.47}$$

et

$$\left\{\begin{array}{l} A_2\,E_2^{n+1} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \\ e_{2,N_2}^{n+1} = e_{1,N_2}^n . \end{array}\right. \tag{3.48}$$

On a ainsi pour $n \geq 1$, $e_{1,j}^n = \gamma_1^n (j-1)\Delta x$ et $e_{2,j}^n = \gamma_2^n (j-N)\Delta x$. Les conditions de Dirichlet en $j = N_1$ et $j = N_2$ donnent

$$\begin{array}{l} \gamma_1^{n+1}(N_1-1)\Delta x = \gamma_2^n (N_1 - N)\Delta x \\ \gamma_2^{n+1}(N_2-N)\Delta x = \gamma_1^n (N_2 - 1)\Delta x \end{array}$$

On a donc

$$\gamma_1^{n+1} = \frac{N_1-N}{N_1-1}\,\gamma_2^n = \frac{N_1-N}{N_1-1}\,\frac{N_2-1}{N_2-N}\,\gamma_1^{n-1} = \frac{N_2-1}{N_1-1}\,\frac{N-N_1}{N-N_2}\,\gamma_1^{n-1} \equiv \beta\gamma_1^{n-1}.$$

La relation est aussi valable pour γ_2^{n+1}. Comme $1 < N_2 \leq N_1 < N$, le terme β est toujours positif et on a: $\beta < 1 \Longleftrightarrow N_2 < N_1$. Pour $N_2 = N_1$, on a $\beta = 1$ et l'algorithme stagne. On a prouvé le

Théorème 3.4.2 *L'algorithme de Schwarz (3.45)-(3.46) converge si et seulement si les sous-domaines se recouvrent ($N_2 < N_1$).*

Remarque 3.4.2 *Le théorème est tout à fait similaire au théorème 3.2.2 du cas continu.*

Remarque 3.4.3 *L'absence de convergence dans le cas sans recouvrement est en accord avec le fait que le problème P'_h est alors mal posé.*

Remarque 3.4.4 *Les systèmes linéaires (3.45) (resp. (3.46)) ne sont pas symétriques à la ligne correspondant à la discrétisation au point x_{N_1-1} (resp. x_{N_2+1}). Ils peuvent être symétrisés en substituant u_{1,N_1}^{n+1} par u_{2,N_1}^n (resp. u_{2,N_2}^{n+1} par u_{1,N_2}^n) dans les lignes correspondantes.*

3.4.3 Reformulation avec des conditions de Robin - Problème P"$_h$

Le but de ce paragraphe est de coupler les systèmes linéaires (3.38)-(3.40) par des conditions de Robin (3.18)-(3.19) discrétisées aux points x_{N_1} et x_{N_2} par (cf. (3.31)):

$$\frac{u_{1,N_1} - u_{1,N_1-1}}{\Delta x} + \alpha u_{1,N_1} - \frac{\Delta x}{2} f(x_{N_1}) = -(\frac{u_{2,N_1} - u_{2,N_1+1}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_1})) + \alpha u_{2,N_1} \tag{3.49}$$

et (attention au changement de direction de la dérivée normale: $n_2 = -n_1$)

$$\frac{u_{2,N_2} - u_{2,N_2+1}}{\Delta x} + \frac{\Delta x}{2} f(x_{N_2}) + \alpha u_{2,N_2} = -(\frac{u_{1,N_2} - u_{1,N_2-1}}{\Delta x} + \frac{\Delta x}{2} f(x_{N_2})) + \alpha u_{1,N_1} \tag{3.50}$$

où α est un nombre strictement positif. Nous désignerons encore par **P"**$_h$ le problème constitué par les équations (3.38)-(3.40)-(3.49)-(3.50) Nous allons montrer que l'unique solution du problème P"$_h$ est la restriction aux sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$ de la solution du problème P$_h$ (3.37) posé sur l'ensemble des inconnues. Ce résultat est valable avec ou sans recouvrement des sous-domaines.

Remarque (Importante) 3.4.1 *Les discrétisations des dérivées normales (3.18)-(3.19) par (3.49)-(3.50) ne font pas appel aux même points suivant que l'on soit dans le sous-domaine* 1 *ou dans le sous-domaine* 2. *On pourrait s'inquiéter des problèmes de consistance avec le problème P_h. Comme le montre le paragraphe* **Existence** *ci-dessous, la dépendance de (3.49) et (3.50) par rapport au second membre f assure l'équivalence des formulations.*

Existence

On construit une solution de P"$_h$ à partir de la solution U du système (3.37) en posant $U_1 = (u_j)_{j\in\mathcal{N}_1}$ et $U_2 = (u_j)_{j\in\mathcal{N}_2}$. Le couple (U_1, U_2) ainsi formé vérifie évidemment les équations (3.38) et (3.40). Vérifions que l'équation (3.49) est satisfaite. Comme $u_{1,N_1} = u_{2,N_1}$ ceci équivaut à montrer que

$$\frac{u_{1,N_1} - u_{1,N_1-1}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_1}) = -\frac{u_{2,N_1} - u_{2,N_1+1}}{\Delta x} + \frac{\Delta x}{2} f(x_{N_1})$$

soit encore par définition de U_1 et U_2

$$\frac{u_{N_1} - u_{N_1-1}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_1}) = -\frac{u_{N_1} - u_{N_1+1}}{\Delta x} + \frac{\Delta x}{2} f(x_{N_1}).$$

En divisant par Δx ceci équivaut à

$$\frac{u_{N_1-1} + 2u_{N_1} - u_{N_1+1}}{(\Delta x)^2} = f(x_{N_1}).$$

Il s'agit du schéma de discrétisation du Laplacien écrit au point x_{N_1}. On montre de même que l'équation (3.50) est satisfaite.

Unicité

Par linéarité, il suffit de considérer le cas homogène $g_0 = g_L = f = 0$. Alors, d'après (3.38) et (3.40), on a

$$U_1 = (b_1(j-1)\Delta x)_{j\in\mathcal{N}_1}$$

et

$$U_2 = (b_2(j-N)\Delta x)_{j\in\mathcal{N}_2}$$

où b_1, $b_2 \in \mathbb{R}$. Les conditions de raccord (3.49)-(3.50) deviennent:

$$b_1 + \alpha(b_1(N_1-1)\Delta x) = b_2 + \alpha(b_2(N_1-N)\Delta x)$$

et

$$-b_2 + \alpha(b_2(N_2 - N)\Delta x) = -b_1 + \alpha(b_1(N_2 - 1)\Delta x).$$

Il s'agit d'un système en b_1, b_2:

$$\left\{ \begin{array}{l} b_1(1 + \alpha(N_1 - 1)\Delta x) - b_2(1 + \alpha(N_1 - N)\Delta x) = 0 \\ b_1(1 - \alpha(N_2 - 1)\Delta x) + b_2(-1 + \alpha(N_2 - N)\Delta x = 0. \end{array} \right. \quad (3.51)$$

Le déterminant de (3.51) vaut

$$\alpha\Delta x(1 - N)(2 + \alpha\Delta x(N_1 - N_2)) = \alpha L\,(2 + \alpha(L_1 - l_2))$$

et est strictement positif et égal au déterminant de (3.20). Le système (3.51) admet donc une solution unique. Il en est alors de même du problème P"$_h$.
On a montré le

Lemme 3.4.3 *Le problème P"$_h$ (Eqs. (3.38)-(3.40)-(3.49)-(3.50)) admet une unique solution. La solution (U_1, U_2) du problème P"$_h$ est composée des restrictions aux sous-ensembles d'indices de la solution U du problème P_h (i.e. $U_1 = U_{|\mathcal{N}_1}$ et $U_2 = U_{|\mathcal{N}_2}$).*

3.4.4 Algorithme parallèle de résolution du problème P"$_h$

Nous considérons une méthode itérative de résolution du problème P"$_h$. On part de U_1^0 et U_2^0 quelconques et on pose

$$\left\{ \begin{array}{l} A_1\,U_1^{n+1} = \begin{pmatrix} g_0 \\ f(x_2) + \frac{g_0}{(\Delta x)^2} \\ f(x_3) \\ \vdots \\ f(x_{N_1-1}) \end{pmatrix}, \\ \dfrac{u_{1,N_1}^{n+1} - u_{1,N_1-1}^{n+1}}{\Delta x} + \alpha u_{1,N_1}^{n+1} - \dfrac{\Delta x}{2} f(x_{N_1}) \\ \qquad\qquad = -(\dfrac{u_{2,N_1}^{n} - u_{2,N_1+1}^{n}}{\Delta x} - \dfrac{\Delta x}{2} f(x_{N_1})) + \alpha u_{2,N_1}^{n} \end{array} \right. \quad (3.52)$$

et

$$\left\{ \begin{array}{l} A_2\,U_2^{n+1} = \begin{pmatrix} f(x_{N_2-1}) \\ \vdots \\ f(x_{N-2}) \\ f(x_{N-1}) + \frac{g_L}{(\Delta x)^2} \\ g_L \end{pmatrix}, \\ \dfrac{u_{2,N_2}^{n+1} - u_{2,N_2+1}^{n+1}}{\Delta x} + \dfrac{\Delta x}{2} f(x_{N_2}) + \alpha u_{2,N_2}^{n+1} \\ \qquad\qquad = -(\dfrac{u_{1,N_2}^{n} - u_{1,N_2-1}^{n}}{\Delta x} + \dfrac{\Delta x}{2} f(x_{N_2})) + \alpha u_{1,N_2}^{n}. \end{array} \right. \quad (3.53)$$

Remarque 3.4.5 *Une solution stationnaire de l'algorithme est solution du problème P"$_h$. Ceci prouve que si l'algorithme converge (voir plus loin), il converge vers l'unique solution de P"$_h$ qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution u du problème P_h. L'algorithme de Schwarz (3.52)-(3.53) sera donc une méthode itérative de résolution du problème P_h.*

Pour étudier la convergence, on pose $E_1^n = (e_{1,j}^n)_{j\in\mathcal{N}_1} = U_1^n - U_1$ et $E_2^n = (e_{2,j}^n)_{j\in\mathcal{N}_2} = U_2^n - U_2$ où (U_1, U_2) est la solution du problème P"$_h$. Par linéarité des équations, l'erreur vérifie

$$\begin{cases} A_1\, E_1^{n+1} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \\ \dfrac{e_{1,N_1}^{n+1} - e_{1,N_1-1}^{n+1}}{\Delta x} + \alpha e_{1,N_1}^{n+1} \\ \qquad\qquad = -\dfrac{e_{2,N_1}^{n} - e_{2,N_1+1}^{n}}{\Delta x} + \alpha e_{2,N_1}^{n} \end{cases} \tag{3.54}$$

et

$$\begin{cases} A_2\, E_2^{n+1} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}, \\ \dfrac{e_{2,N_2}^{n+1} - e_{2,N_2+1}^{n+1}}{\Delta x} + \alpha e_{2,N_2}^{n+1} \\ \qquad\qquad = -\dfrac{e_{1,N_2}^{n} - e_{1,N_2-1}^{n}}{\Delta x} + \alpha e_{1,N_2}^{n}. \end{cases} \tag{3.55}$$

On a ainsi pour $n \geq 1$, $e_{1,j}^n = \gamma_1^n (j-1)\Delta x$ et $e_{2,j}^n = \gamma_2^n (j-N)\Delta x$. Les conditions de Robin en $j = N_1$ et $j = N_2$ donnent

$$\gamma_1^{n+1}(1 + \alpha(N_1 - 1)\Delta x) = \gamma_2^n (1 + \alpha(N_1 - N)\Delta x)$$
$$\gamma_2^{n+1}(-1 + \alpha(N_2 - N)\Delta x) = \gamma_1^n (-1 + \alpha(N_2 - 1)\Delta x)$$

On a donc

$$\gamma_1^{n+1} = \frac{1 + \alpha(N_1 - N)\Delta x}{1 + \alpha(N_1 - 1)\Delta x}\gamma_2^n = \frac{1 + \alpha(N_1 - N)\Delta x}{1 + \alpha(N_1 - 1)\Delta x}\,\frac{-1 + \alpha(N_2 - 1)\Delta x}{-1 + \alpha(N_2 - N)\Delta x} \equiv \beta \gamma_1^{n-1}$$

où

$$\beta = \frac{1 + \alpha(N_1 - N)\Delta x}{-1 + \alpha(N_2 - N)\Delta x}\,\frac{-1 + \alpha(N_2 - 1)\Delta x}{1 + \alpha(N_1 - 1)\Delta x}.$$

La relation est aussi valable pour γ_2^{n+1}. Le paramètre α étant positif, on peut montrer qu'avec ou sans recouvrement des sous-domaines,

$$-1 < \beta < 1.$$

On a ainsi prouvé le

Théorème 3.4.4 *Pour $\alpha > 0$, l'algorithme (3.52)-(3.53) converge.*

Remarque 3.4.6 *Contrairement à l'algorithme de Schwarz, le recouvrement des sous-domaines n'est pas nécessaire pour assurer la convergence de l'algorithme.*

Remarque 3.4.7 *Il est possible d'annuler β et donc d'avoir une convergence en deux itérations en choisissant $\alpha = \frac{1}{(N_2-1)\Delta x}$ ou bien $\alpha = \frac{1}{(N-N_1)\Delta x}$.*

Chapitre 4

Equation de Laplace en dimension 2

4.1 Aspect mathématique

4.1.1 Le problème aux limites sur un rectangle

On note $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. On considère l'équation de Laplace

$$-\Delta u = 0 \tag{4.1}$$

sur le rectangle: $\Omega = [0, L_x] \times [0, L_y]$ et où u est une fonction scalaire de la variable d'espace $(x, y) \in \Omega$. La frontière de Ω se décompose en quatre segments: $\Gamma_b =]0, L_x[\times\{0\}$, $\Gamma_d = \{L_x\}\times]0, L_y[$, $\Gamma_h =]0, L_x[\times\{L_y\}$ et $\Gamma_g = \{0\}\times]0, L_y[$. On étudiera différents jeux de conditions aux limites:

$$u = 0 \text{ sur } \Gamma_b \cup \Gamma_h,\ \ u = g_g \text{ sur } \Gamma_g \text{ et } u = g_d \text{ sur } \Gamma_d \tag{4.2}$$

ou bien

$$u = 0 \text{ sur } \Gamma_b \cup \Gamma_h,\ \ u = g_g \text{ sur } \Gamma_g \text{ et } (\frac{\partial u}{\partial n} + \alpha u) = g_d \text{ sur } \Gamma_d, \tag{4.3}$$

où α est une constante positive ou nulle, n est la normale extérieure au domaine Ω et g_g, g_d sont des fonctions scalaires. Nous désignerons par **P1r** le problème (4.1)-(4.2) et par **P2r** le problème (4.1)-(4.3). Nous allons étudier formellement ces problèmes à l'aide de l'analyse de Fourier qui nous sera utile par la suite pour l'analyse des méthodes de décomposition de domaine. Il existe des méthodes d'analyse mathématique plus générales [1] dont nous ne nous servirons pas ici.

Nous introduisons la famille de fonctions

$$z_n = \sqrt{\frac{2}{L_y}} \sin(\frac{n\pi y}{L_y}), \quad n \geq 1 \tag{4.4}$$

qui sont telles que $z_n(0) = z_n(L_y) = 0$. On sait que les fonctions z_n forment une base orthogonale de $L^2(]0, L_y[)$ et une base orthonormale de $H^1(]0, L_y[)$. On peut écrire $u(x, y)$ sous la forme

$$u(x, y) = \sum_{n=1}^{\infty} u_n(x) z_n(y) \tag{4.5}$$

1. voir le polycopié [LL]

qui satisfait les conditions de Dirichlet homogènes sur Γ_b et Γ_h.
L'équation (4.1) se traduit par:

$$\sum_{n=1}^{\infty} \frac{d^2 u_n}{dx^2} z_n + \sum_{n=1}^{\infty} u_n \left(-\frac{n^2 \pi^2}{{L_y}^2}\right) z_n = 0 \tag{4.6}$$

soit

$$\sum_{n=1}^{\infty} \left(\frac{d^2 u_n}{dx^2} - \frac{n^2 \pi^2}{{L_y}^2} u_n\right) z_n = 0 \tag{4.7}$$

d'où

$$\frac{d^2 u_n}{dx^2} - \frac{n^2 \pi^2}{{L_y}^2} u_n = 0, \quad \forall n \geq 1. \tag{4.8}$$

Pour alléger les notations, on pose

$$k_n = \frac{n\pi}{L_y}, \qquad n \geq 1. \tag{4.9}$$

La solution générale de (4.8) est de la forme

$$u_n(x) = \delta_n e^{-k_n x} + \beta_n e^{k_n (x - L_x)}. \tag{4.10}$$

Les coefficients δ_n et β_n seront déterminés par les conditions aux limites sur Γ_g et Γ_d qui sont différentes pour les problèmes P1r et P2r. On introduit les décompositions de g_g et g_d en série de Fourier

$$g_g = \sum_{n=1}^{\infty} g_{gn}\, z_n(y) \text{ et } g_d = \sum_{n=1}^{\infty} g_{dn}\, z_n(y) \tag{4.11}$$

où

$$g_{gn} = \int_0^{L_y} g_g(y)\, z_n(y)\, dy \text{ et } g_{dn} = \int_0^{L_y} g_d(y)\, z_n(y)\, dy.$$

Etude du problème P1r

La condition à la limite (4.2) sur Γ_g s'écrit

$$\sum_{n=1}^{\infty} u_n(0)\, z_n(y) = g_g(y) = \sum_{n=1}^{\infty} g_{gn}\, z_n(y)$$

soit encore

$$u_n(0) = g_{gn}, \quad \forall n \geq 1.$$

De même, la condition à la limite sur Γ_d donne

$$g_{dn} = u_n(L_x).$$

En utilisant (4.10), les conditions aux limites sur Γ_g et Γ_d se traduisent par un système en δ_n et β_n:

$$\begin{aligned} \delta_n + \beta_n e^{-k_n L_x} &= g_{gn} \\ \delta_n e^{-k_n L_x} + \beta_n &= g_{dn}. \end{aligned} \tag{4.12}$$

Le déterminant du système linéaire (4.12) vaut $1 - e^{-2k_n L_x}$ et est différent de zéro. On a

$$\begin{aligned} \delta_n &= \frac{g_{gn} - g_{dn} e^{-k_n L_x}}{1 - e^{-2k_n L_x}} \\ \beta_n &= \frac{g_{dn} - g_{gn} e^{-k_n L_x}}{1 - e^{-2k_n L_x}}. \end{aligned}$$

Ainsi, le problème P1r admet une solution unique donnée par

$$u(x,y) = \sum_{n=1}^{\infty} \left[\frac{g_{gn}(e^{-k_n x} - e^{k_n(x-2L_x)}) + g_{dn}(e^{k_n(x-L_x)} - e^{-k_n(x+L_x)})}{1 - e^{-2k_n L_x}} \right] z_n(y). \tag{4.13}$$

Etude du problème P2r

Par rapport au problème P1r, seule la condition à la limite sur Γ_d change. D'après (4.3), on a

$$\sum_{n=1}^{\infty} \frac{d\,u_n}{dx}(L_x)\, z_n(y) + \alpha \sum_{n=1}^{\infty} u_n(L_x)\, z_n(y) = g_d(y),$$

soit

$$\sum_{n=1}^{\infty} (\frac{d\,u_n}{dx}(L_x) + \alpha u_n(L_x))\, z_n(y) = g_d(y)$$

ou encore en utilisant la décomposition en série de Fourier de g_d (4.11):

$$\sum_{n=1}^{\infty} (\frac{d\,u_n}{dx}(L_x) + \alpha u_n(L_x))\, z_n(y) = \sum_{n=1}^{\infty} g_{dn}\, z_n(y),$$

soit pour tout $n \geq 1$:

$$\frac{d\,u_n}{dx}(L_x) + \alpha u_n(L_x) = g_{dn}. \tag{4.14}$$

La première ligne de (4.12) et les équations (4.14) et (4.10) donnent

$$\begin{array}{l} \delta_n + \beta_n e^{-k_n L_x} = g_{gn} \\ \delta_n(-k_n + \alpha) e^{-k_n L_x} + \beta_n(k_n + \alpha) = g_{dn}. \end{array} \tag{4.15}$$

Le déterminant de (4.15) vaut $(k_n + \alpha) - (-k_n + \alpha)e^{-2k_n L_x}$ et est strictement supérieur à zéro car α, $k_n > 0$. On a

$$\begin{array}{l} \delta_n = \dfrac{(k_n + \alpha) g_{gn} - g_{dn} e^{-k_n L_x}}{k_n + \alpha - (-k_n + \alpha) e^{-2k_n L_x}} \\ \beta_n = \dfrac{g_{dn} - g_{gn}(-k_n + \alpha) e^{-k_n L_x}}{k_n + \alpha - (-k_n + \alpha) e^{-2k_n L_x}}. \end{array}$$

Ainsi, le problème P2r admet une solution unique donnée par

$$u(x,y) = \sum_{n=1}^{\infty} \left[\frac{g_{gn}(e^{-k_n x} - \frac{-k_n+\alpha}{k_n+\alpha} e^{k_n(x-2L_x)}) + \frac{g_{dn}}{k_n+\alpha}(e^{k_n(x-L_x)} - e^{-k_n(x+L_x)})}{1 - \frac{-k_n+\alpha}{k_n+\alpha} e^{-2k_n L_x}} \right] z_n(y). \tag{4.16}$$

4.1.2 Le problème aux limites sur une bande semi-infinie

Les formules (4.13) et (4.16) se simplifient notablement si on considère des équations de Laplace posées dans des domaines infinis dans une direction d'espace. On considère l'équation de Laplace sur une bande semi-infinie

$$-\Delta u = 0 \text{ dans } \Omega =]0, \infty[\times]0, L_y[. \tag{4.17}$$

La frontière de Ω se décompose en trois parties: $\Gamma_b =]0, \infty[\times\{0\}$, $\Gamma_h =]0, \infty[\times\{L_y\}$ et $\Gamma_g = \{0\}\times]0, L_y[$. On étudiera différents jeux de conditions aux limites:

$$u = 0 \text{ sur } \Gamma_b \cup \Gamma_h, \quad \text{et } u = g_g \text{ sur } \Gamma_g \tag{4.18}$$

ou bien

$$u = 0 \text{ sur } \Gamma_b \cup \Gamma_h, \quad \text{et } (\frac{\partial u}{\partial n} + \alpha u) = g_g \text{ sur } \Gamma_g, \tag{4.19}$$

où α est une constante positive ou nulle, n est la normale extérieure au domaine Ω et g_g, g_d sont des fonctions scalaires. Nous désignerons par **P1** le problème (4.17)-(4.18) et par **P2** le problème (4.17)-(4.19). Les formules explicites des solutions s'obtiennent à partir des formules (4.13) et (4.16). En faisant tendre L_x vers l'infini dans (4.13), on obtient la formule suivante pour l'unique solution du problème P1:

$$u(x,y) = \sum_{n=1}^{\infty} g_{gn} e^{-k_n x} z_n(y). \tag{4.20}$$

En faisant le changement de variable $x \longrightarrow -x$ et en faisant tendre L_x vers l'infini dans la formule (4.16), on obtient la formule suivante pour l'unique solution du problème P2:

$$u(x,y) = \sum_{n=1}^{\infty} \frac{g_{gn}}{k_n + \alpha} e^{-k_n x} z_n(y). \tag{4.21}$$

Les fonctions z_n et les coefficients g_{gn} et k_n sont définis, respectivement, par les formules (4.4), (4.11) et (4.9) ci-dessus.

Remarque 4.1.1 *Le cas de l'équation (4.17) avec un second membre f peut se traiter de la même manière en utilisant la formule de Duhamel. Les expressions ne sont pas données car elles ne seront pas utilisées pour étudier les méthodes de décomposition de domaine. Par la suite, on admettra que les problèmes aux limites sont bien posés. La théorie complète se trouve exposée par exemple dans le polycopié [LL].*

4.2 Méthodes de décomposition de domaine - Cas continu

Nous considérons l'équation

$$-\Delta u = f \tag{4.22}$$

où u et f sont des fonctions scalaires de la variable d'espace $(x,y) \in \mathbb{R}\times]0, L_y[$ avec des conditions aux limites de Dirichlet homogènes:

$$u = 0 \text{ sur } \mathbb{R} \times \{0\} \cup \mathbb{R} \times \{L_y\} \tag{4.23}$$

La bande infinie $\mathbb{R}\times]0, L_y[$ est décomposée en deux sous-domaines $\Omega_1 = [-\infty, L_1[\times]0, L_y[$ et $\Omega_2 = [l_2, \infty[\times]0, L_y[$, $0 < l_2 \leq L_1 < L$. Nous désignerons par **P** le problème défini par (4.22)-(4.23). Nous allons réécrire P de deux manières différentes sous la forme de deux problèmes couplés posés dans les sous-domaines Ω_1 et Ω_2. Puis, nous étudierons une méthode itérative de résolution de ces systèmes couplés adaptée aux architectures parallèles.

On désigne par u_1 la restriction de u au domaine Ω_1 et par u_2 sa restriction au domaine Ω_2. Le couple (u_1, u_2) vérifie

$$\left\{ \begin{array}{l} -\Delta u_1 = f \text{ dans } \Omega_1 \\ u_1 = 0 \text{ sur }]-\infty, L_1[\times\{0\}\cup]-\infty, L_1[\times\{L_y\} \end{array} \right. \tag{4.24}$$

$$\left\{ \begin{array}{l} -\Delta u_2 = f \text{ dans } \Omega_2 \\ u_2 = 0 \text{ sur }]l_2, \infty[\times\{0\}\cup]-\infty, L_1[\times\{L_y\} \end{array} \right. \tag{4.25}$$

D'après (4.25), la forme générale de u_2 est donnée par

$$u_2 = u_{|\Omega_2} + w_2$$

où w_2 est une fonction dont le laplacien est nul, qui s'annule sur $]l_2, \infty[\times\{0\} \cup]l_2, \infty[\times\{L_y\}$ mais prend une valeur quelconque sur $\{l_2\}\times]0, L_y[$. La fonction w_2 est donc solution d'un problème du type P1 avec la condition de Dirichlet tautologique $w_2(l_2, y) = w_2(l_2, y)$. D'après le paragraphe 4.1.2 (et à une translation près), la forme générale de w_2 est

$$w_2(x, y) = \sum_{n=1}^{\infty} g_{2n} e^{-k_n(x-l_2)} z_n(y) \tag{4.26}$$

où $(g_{2n})_{n\geq 1}$ est une suite arbitraire de coefficients de Fourier. De même, d'après (4.24), la forme générale de u_1 est donnée par

$$u_1 = u_{|\Omega_1} + w_1$$

où w_1 est une fonction dont le laplacien est nul, qui s'annule sur $]-\infty, L_1[\times\{0\} \cup]-\infty, L_1[\times\{L_y\}$ mais prend une valeur quelconque sur $\{L_1\}\times]0, L_y[$. La fonction w_1 est donc solution d'un problème du type P1 avec la condition de Dirichlet tautologique $w_1(L_1, y) = w_1(L_1, y)$. D'après le paragraphe 4.1.2, la forme générale de w_1 est (après un changement de variables)

$$w_1(x, y) = \sum_{n=1}^{\infty} g_{1n} e^{k_n(x-L_1)} z_n(y) \tag{4.27}$$

où $(g_{1n})_{n\geq 1}$ est une suite arbitraire de coefficients de Fourier.

Remarque (Importante) 4.2.1 *Les termes en x intervenant dans (4.26) et (4.27) sont exponentiellement décroissants.*

Ainsi, les équations (4.24)-(4.25) ne suffisent pas à déterminer de manière unique le couple (u_1, u_2). Dans les paragraphes qui suivent, nous ajoutons des conditions de raccord aux interfaces (dites aussi conditions d'interface) $x = L_1$ et $x = l_2$ pour que le couple (u_1, u_2) soit déterminé de manière unique et soit égal aux restrictions de u aux sous-domaines Ω_1 et Ω_2. Pour chaque jeu de conditions d'interfaces, on propose une méthode itérative de résolution adaptée aux architectures parallèles.

4.2.1 Reformulation avec des conditions de Dirichlet - Problème P'

Nous ajoutons des conditions de raccord de type Dirichlet:

$$u_1(L_1, y) = u_2(L_1, y) \quad \forall y \in]0, L_y[\tag{4.28}$$

et

$$u_2(l_2, y) = u_1(l_2, y) \quad \forall y \in]0, L_y[. \tag{4.29}$$

Nous désignerons par **P'** le problème (4.24)-(4.25)-(4.28)-(4.29). Nous allons montrer que ce problème admet un couple solution unique (u_1, u_2) si et seulement si les sous-domaines se recouvrent $(L_1 > l_2)$.

Existence

La solution u du problème P étant continuement dérivable, le couple (u_1, u_2), où $u_1 = u_{|\Omega_1}$ et $u_2 = u_{|\Omega_2}$, est solution du problème P'.

Remarquons que s'il y unicité, la solution de P' est la restriction de u aux sous-domaines.

Unicité

Par linéarité des équations, il suffit de considérer les équations homogènes $f = 0$. D'après (4.24) et (4.25), u_1 et u_2 sont donc solutions d'un problème du type P1 avec les conditions de Dirichlet tautologiques $u_1(L_1,y) = u_1(L_1,y)$ et $u_2(l_2,y) = u_2(l_2,y)$. Ainsi, on a

$$u_1(x,y) = \sum_{n=1}^{\infty} g_{1n} e^{k_n(x-L_1)} z_n(y)$$

et

$$u_2(x,y) = \sum_{n=1}^{\infty} g_{2n} e^{-k_n(x-l_2)} z_n(y)$$

où $(g_{1n})_{n\geq 1}$ et $(g_{2n})_{n\geq 1}$ sont des suites arbitraires de coefficients de Fourier.
La condition de raccord (4.28) devient:

$$\sum_{n=1}^{\infty} g_{1n}\, z_n(y) = \sum_{n=1}^{\infty} g_{2n} e^{-k_n(L_1-l_2)} z_n(y)$$

soit encore

$$\sum_{n=1}^{\infty} (g_{1n} - g_{2n} e^{-k_n(L_1-l_2)})\, z_n(y) = 0,$$

d'où

$$g_{1n} - g_{2n} e^{-k_n(L_1-l_2)} = 0, \quad \forall n \geq 1. \tag{4.30}$$

De même, la condition de raccord (4.29) devient

$$-g_{1n} e^{k_n(l_2-L_1)} + g_{2n} = 0, \quad \forall n \geq 1. \tag{4.31}$$

Pour chaque $n \geq 1$, le système linéaire en (g_{1n}, g_{2n}) (4.30)-(4.31) a un déterminant qui vaut $1 - e^{-2k_n(L_1-l_2)}$.
Il est non nul ssi les sous-domaines se recouvrent, $l_2 < L_1$. Alors, on a $g_{1n} = g_{2n} = 0$ pour tout $n \geq 1$ et donc $u_1 = u_2 = 0$. Ceci prouve l'unicité.
Dans le cas où les sous-domaines sont adjacents ($l_2 = L_1$), le déterminant de (4.30)-(4.31) s'annule et le système linéaire (4.30)-(4.31) admet une infinité de solutions $g_{1n} = g_{2n}, g_{2n} = g_{2n}$. Il en est alors de même du problème P' qui admet une infinité de solutions. On a ainsi montré le
Lemme 4.2.1 *Le problème P' (Eqs. (4.24)-(4.25)-(4.28)-(4.29)) admet une unique solution ssi les sous-domaines se recouvrent. Dans ce cas, la solution (u_1,u_2) de P' est composée des restrictions, aux sous-domaines, de la solution u du problème P (i.e. $u_1 = u$ sur Ω_1 et $u_2 = u$ sur Ω_2).*

4.2.2 Algorithme parallèle de résolution du problème P' (Algorithme de Schwarz)

Nous considérons une méthode itérative de résolution du problème P'. On part de u_1^0 et u_2^0 quelconques et on pose

$$\left\{\begin{array}{l} -\Delta u_1^{n+1} = f \text{ dans } \Omega_1 \\ u_1^{n+1} = 0 \text{ sur }]-\infty, L_1[\times\{0\} \cup]-\infty, L_1[\times\{L_y\} \\ u_1^{n+1}(L_1,y) = u_2^n(L_1,y) \end{array}\right. \tag{4.32}$$

$$\left\{\begin{array}{l} -\Delta u_2^{n+1} = f \text{ dans } \Omega_2 \\ u_2^{n+1} = 0 \text{ sur }]l_2,\infty[\times\{0\} \cup]l_2,\infty[\times\{L_y\} \\ u_2^{n+1}(l_2,y) = u_1^n(l_2,y) \end{array}\right. \tag{4.33}$$

Les problèmes aux limites définissant l'algorithme sont du type P1 et donc bien posés.
Remarque 4.2.1 *Une solution stationnaire de l'algorithme est solution du problème P'. Ceci prouve que si l'algorithme converge, il converge vers l'unique solution (quand les sous-domaines se chevauchent) de P' qui est la restriction aux*

sous-domaines Ω_1 et Ω_2 de la solution u du problème P. L'algorithme (4.32)-(4.33) sera donc une méthode itérative de résolution du problème P.

Pour étudier la convergence, on pose $e_1^n = u_1^n - u_1$ et $e_2^n = u_2^n - u_2$ où (u_1, u_2) est la solution du problème P'. Par linéarité des équations, l'erreur (e_1^n, e_2^n) vérifie

$$\left\{ \begin{array}{l} -\Delta e_1^{n+1} = 0 \text{ dans } \Omega_1 \\ e_1^{n+1} = 0 \text{ sur }]-\infty, L_1[\times\{0\}\cup]-\infty, L_1[\times\{L_y\} \\ e_1^{n+1}(L_1, y) = e_2^n(L_1, y) \end{array} \right. \tag{4.34}$$

$$\left\{ \begin{array}{l} -\Delta e_2^{n+1} = 0 \text{ dans } \Omega_2 \\ e_2^{n+1} = 0 \text{ sur }]l_2, \infty[\times\{0\}\cup]l_2, \infty[\times\{L_y\} \\ e_2^{n+1}(l_2, y) = e_1^n(l_2, y) \end{array} \right. \tag{4.35}$$

On a ainsi pour $n \geq 1$,

$$e_1^n(x, y) = \sum_{m=1}^{\infty} g_{1m}^n e^{k_m(x-L_1)} z_m(y)$$

et

$$e_2^n(x, y) = \sum_{m=1}^{\infty} g_{2m}^n e^{-k_m(x-l_2)} z_m(y).$$

Les conditions de Dirichlet en $x = L_1$ et $x = l_2$ donnent

$$\sum_{m=1}^{\infty} g_{1m}^{n+1} z_m(y) = \sum_{m=1}^{\infty} g_{2m}^n e^{-k_m(L_1-l_2)} z_m(y)$$

et

$$\sum_{m=1}^{\infty} g_{2m}^{n+1} z_m(y) = \sum_{m=1}^{\infty} g_{1m}^n e^{k_m(l_2-L_1)} z_m(y).$$

soit encore pour tout $n \geq 1$,

$$\begin{array}{l} g_{1m}^{n+1} = g_{2m}^n e^{-k_m(L_1-l_2)}, \\ g_{2m}^{n+1} = g_{1m}^n e^{k_m(l_2-L_1)}. \end{array}$$

On a donc pour tout $m \geq 1$ et $n \geq 2$:

$$g_{1m}^{n+1} = g_{2m}^n e^{-k_m(L_1-l_2)} = g_{1m}^{n-1} e^{-2k_m(L_1-l_2)} \equiv \beta_m g_{1m}^{n-1}.$$

La relation est aussi valable pour g_{2m}^{n+1}. Il faut distinguer deux cas suivant le chevauchement des sous-domaines. En effet, pour $l_2 = L_1$, on a $\beta_m = 1$ et l'algorithme stagne. Par contre, pour $l_2 < L_1$, on a une majoration uniforme de β_m: $\beta_m \leq \beta_1 < 1$. Par application du théorème de Parseval-Plancherel, on a

$$\begin{aligned} \|e_1^{2n+1}\|^2_{L_2(\Omega_1)} &= \int_{-\infty}^{L_1} \sum_{m=1}^{\infty} e^{2k_m(x-L_1)} (g_{1m}^{2n+1})^2 = \sum_{m=1}^{\infty} \frac{(g_{1m}^{2n+1})^2}{2k_m} \\ &= \sum_{m=1}^{\infty} \beta_m^n \frac{(g_{1m}^1)^2}{2k_m} \leq \beta_1^n \sum_{m=1}^{\infty} \frac{(g_{1m}^1)^2}{2k_m} \leq \beta_1^n \|e_1^1\|^2_{L^2(]0,L_y[)}. \end{aligned}$$

La norme L^2 de l'erreur converge géométriquement vers zéro.

On a prouvé le

Théorème 4.2.2 *L'algorithme de Schwarz (4.32)-(4.33) converge si et seulement si les sous-domaines se recouvrent ($l_2 < L_1$).*

Remarque 4.2.2 *L'absence de convergence dans le cas sans recouvrement est en accord avec le fait que le problème P' est alors mal posé.*

Remarque 4.2.3 *Plus le nombre de Fourier $k_m = \frac{m\pi}{L_y}$ est grand, plus le facteur de convergence β_m est petit. Rappelons que cela correspond à la composante de l'erreur selon $z_m = \sin(k_m y)$. Les composantes les plus oscillantes de l'erreur le long des interfaces $x = L_1$ et $x = l_2$ sont ainsi très rapidement atténuées par l'algorithme.*

Remarque 4.2.4 *Les conclusions précédent valables pour un domaine borné dans la direction x.*

4.2.3 Reformulation avec des conditions de Robin - Problème P"

Pour fermer le système d'équations (4.24)-(4.25), nous ajoutons des conditions de raccord de type Robin:

$$(\frac{\partial u_1}{\partial n_1} + \alpha u_1)(L_1, y) = (-\frac{\partial u_2}{\partial n_2} + \alpha u_2)(L_1, y) \quad \forall y \in]0, L_y[\tag{4.36}$$

et

$$(\frac{\partial u_2}{\partial n_2} + \alpha u_2)(l_2) = (-\frac{\partial u_1}{\partial n_1} + \alpha u_1)(l_2) \quad \forall y \in]0, L_y[\tag{4.37}$$

où α est une constante strictement positive, n_1 (resp. n_2) désigne la normale extérieure au domaine Ω_1 (resp. Ω_2). Les signes $-$ devant les dérivées normales viennent du fait que $n_2 = -n_1$. Nous désignerons par **P"** le problème (4.24)-(4.36)-(4.25)-(4.37). Nous allons montrer que ce problème admet un couple solution unique (u_1, u_2) avec ou sans recouvrement des sous-domaines.

Existence

La solution u du problème P étant continuement dérivable, le couple (u_1, u_2), où $u_1 = u_{|\Omega_1}$ et $u_2 = u_{|\Omega_2}$, est solution du problème P". Remarquons que s'il y unicité la solution de P" est la restriction de u aux sous-domaines.

Unicité

Par linéarité des équations, il suffit de considérer les équations homogènes $f = 0$. D'après (4.24) et (4.25), u_1 et u_2 sont donc solutions d'un problème du type P1 avec les conditions de Dirichlet tautologiques $u_1(L_1, y) = u_1(L_1, y)$ et $u_2(l_2, y) = u_2(l_2, y)$. Ainsi, on a

$$u_1(x, y) = \sum_{n=1}^{\infty} g_{1n} e^{k_n(x-L_1)} z_n(y)$$

et

$$u_2(x, y) = \sum_{n=1}^{\infty} g_{2n} e^{-k_n(x-l_2)} z_n(y)$$

où $(g_{1n})_{n\geq 1}$ et $(g_{2n})_{n\geq 1}$ sont des suites arbitraires de coefficients de Fourier.
La condition de raccord (4.36) devient:

$$\sum_{n=1}^{\infty} g_{1n}(k_n + \alpha)\, z_n(y) = \sum_{n=1}^{\infty} g_{2n}(-k_n + \alpha) e^{-k_n(L_1-l_2)}\, z_n(y)$$

soit encore

$$\sum_{n=1}^{\infty} (g_{1n}(k_n + \alpha) - g_{2n}(-k_n + \alpha) e^{-k_n(L_1-l_2)})\, z_n(y) = 0,$$

d'où

$$(k_n + \alpha) g_{1n} - (-k_n + \alpha) g_{2n} e^{-k_n(L_1-l_2)} = 0, \quad \forall n \geq 1. \tag{4.38}$$

De même, la condition de raccord (4.37) devient

$$-g_{1n}(-k_n+\alpha)e^{k_n(l_2-L_1)}+g_{2n}(k_n+\alpha)=0, \quad \forall n\geq 1. \tag{4.39}$$

Pour chaque $n \geq 1$, le déterminant du système linéaire en (g_{1n}, g_{2n}) (4.38)-(4.39) vaut $(k_n+\alpha)^2-(-k_n+\alpha)^2e^{-2k_n(L_1-l_2)}$. et est strictement positif. Le système (4.38)-(4.39) admet comme unique solution la solution nulle. Il en est alors de même du problème P". On a ainsi montré le

Lemme 4.2.3 *Le problème P" (Eqs. (4.24)-(4.36)-(4.25)-(4.37)) admet une unique solution. La solution (u_1, u_2) de P" est composée des restrictions, aux sous-domaines, de la solution u du problème P (i.e. $u_1 = u$ sur Ω_1 et $u_2 = u$ sur Ω_2).*

Remarque 4.2.5 *Il est important de noter que le problème P", contrairement au problème P', admet une solution unique même quand les sous-domaines ne se recouvrent pas.*

4.2.4 Algorithme parallèle de résolution du problème P"

Nous considérons une méthode itérative de résolution du problème P". On part de u_1^0 et u_2^0 quelconques et on pose

$$\left\{\begin{array}{l} -\Delta u_1^{n+1} = f \text{ dans } \Omega_1 \\ u_1^{n+1} = 0 \text{ sur }]-\infty, L_1[\times\{0\}\cup]-\infty, L_1[\times\{L_y\} \\ (\dfrac{\partial u_1^{n+1}}{\partial n_1}+\alpha u_1^{n+1})(L_1,y) = (-\dfrac{\partial u_2^n}{\partial n_2}+\alpha u_2^n)(L_1,y) \quad \forall y\in]0,L_y[\end{array}\right. \tag{4.40}$$

$$\left\{\begin{array}{l} -\dfrac{\partial^2 u_2^{n+1}}{\partial x^2} = f \text{ dans } \Omega_2 \\ u_2^{n+1} = 0 \text{ sur }]l_2, \infty[\times\{0\}\cup]l_2, \infty[\times\{L_y\} \\ (\dfrac{\partial u_2^{n+1}}{\partial n_2}+\alpha u_2^{n+1})(l_2,y) = (-\dfrac{\partial u_1^n}{\partial n_1}+\alpha u_1^n)(l_2,y) \quad \forall y\in]0,L_y[\end{array}\right. \tag{4.41}$$

Les problèmes aux limites définissant l'algorithme sont du type P2 et donc bien posés.

Remarque 4.2.6 *Une solution stationnaire de l'algorithme est solution du problème P". Ceci prouve que si l'algorithme converge (voir plus loin), il converge vers l'unique solution de P" qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution u du problème P. L'algorithme (4.40)-(4.41) sera donc une méthode itérative de résolution du problème P.*

Pour étudier la convergence, on pose $e_1^n = u_1^n - u_1$ et $e_2^n = u_2^n - u_2$ où (u_1, u_2) est la solution du problème P". L'erreur (e_1^n, e_2^n) vérifie par linéarité des équations

$$\left\{\begin{array}{l} -\Delta e_1^{n+1} = 0 \text{ dans } \Omega_1 \\ e_1^{n+1} = 0 \text{ sur }]-\infty, L_1[\times\{0\}\cup]-\infty, L_1[\times\{L_y\} \\ (\dfrac{\partial e_1^{n+1}}{\partial n_1}+\alpha e_1^{n+1})(L_1,y) = (-\dfrac{\partial e_2^n}{\partial n_2}+\alpha e_2^n)(L_1,y) \quad \forall y\in]0,L_y[\end{array}\right. \tag{4.42}$$

$$\left\{\begin{array}{l} -\Delta e_2^{n+1} = 0 \text{ dans } \Omega_2 \\ e_2^{n+1} = 0 \text{ sur }]l_2, \infty[\times\{0\}\cup]l_2, \infty[\times\{L_y\} \\ (\dfrac{\partial e_2^{n+1}}{\partial n_2}+\alpha e_2^{n+1})(l_2,y) = (-\dfrac{\partial e_1^n}{\partial n_1}+\alpha e_1^n)(l_2,y) \quad \forall y\in]0,L_y[\end{array}\right. \tag{4.43}$$

On a ainsi pour $n \geq 1$,

$$e_1^n(x,y) = \sum_{m=1}^{\infty} g_{1m}^n e^{k_m(x-L_1)} z_m(y)$$

et

$$e_2^n(x,y) = \sum_{m=1}^{\infty} g_{2m}^n e^{-k_m(x-l_2)} z_m(y).$$

Les conditions de Robin en $x = L_1$ et $x = l_2$ donnent

$$\sum_{m=1}^{\infty} g_{1m}^{n+1}(k_m+\alpha)\, z_m(y) = \sum_{m=1}^{\infty} g_{2m}^n(-k_m+\alpha)e^{-k_m(L_1-l_2)}\, z_m(y)$$

et

$$\sum_{m=1}^{\infty} g_{2m}^{n+1}(k_m+\alpha)\, z_m(y) = \sum_{m=1}^{\infty} g_{1m}^n(-k_m+\alpha)e^{k_m(l_2-L_1)}\, z_m(y).$$

soit encore pour tout $n \geq 1$,

$$\begin{aligned} g_{1m}^{n+1}(k_m+\alpha) &= g_{2m}^n(-k_m+\alpha)e^{-k_m(L_1-l_2)}, \\ g_{2m}^{n+1}(k_m+\alpha) &= g_{1m}^n(-k_m+\alpha)e^{k_m(l_2-L_1)}. \end{aligned}$$

On a donc pour tout $m \geq 1$ et $n \geq 2$:

$$g_{1m}^{n+1} = g_{2m}^n \frac{-k_m+\alpha}{k_m+\alpha} e^{-k_m(L_1-l_2)} = g_{1m}^{n-1}\left(\frac{-k_m+\alpha}{k_m+\alpha}\right)^2 e^{-2k_m(L_1-l_2)} \equiv \beta_m g_{1m}^{n-1}.$$

On ne considère que l'erreur dans le domaine Ω_1. L'erreur dans le domaine Ω_2 se traite de la même manière. On a ainsi $g_{1m}^{2n+1} = \beta_m^n g_{1m}^1$. Comme on a α, $k_m > 0$, on voit que: $\beta_m < e^{-2k_m(L_1-l_2)} < 1$. Toutefois, en l'absence de recouvrement, β_m tend vers 1 quand m tend vers l'infini. Ceci indique qu'il faut distinguer deux cas.
Cas n^{0}1 Les sous-domaines se recouvrent.
On a une majoration uniforme de β_m: $\beta_m \leq \beta_1 < 1$. Par application du théorème de Parseval-Plancherel, on a

$$\begin{aligned} \|e_1^{2n+1}\|^2_{L_2(\Omega_1)} &= \int_{-\infty}^{L_1} \sum_{m=1}^{\infty} e^{2k_m(x-L_1)}(g_{1m}^{2n+1})^2 = \sum_{m=1}^{\infty} \frac{(g_{1m}^{2n+1})^2}{2k_m} \\ &= \sum_{m=1}^{\infty} \beta_m^n \frac{(g_{1m}^1)^2}{2k_m} \leq \beta_1^n \sum_{m=1}^{\infty} \frac{(g_{1m}^1)^2}{2k_m} \leq \beta_1^n \|e_1^1\|^2_{L^2(]0,L_y[)}. \end{aligned}$$

La norme L^2 de l'erreur converge géométriquement vers zéro.
Cas n^{0}2 Les sous-domaines ne se recouvrent pas.
La formule suivante reste valable:

$$\|e_1^{2n+1}\|^2_{L_2(\Omega_1)} = \sum_{m=1}^{\infty} \beta_m^n \frac{(g_{1m}^1)^2}{2k_m}$$

Rappelons que $\beta_m < 1$ pour tout $m \geq 1$. Chaque terme de la série $(\beta_m^n \frac{(g_{1m}^1)^2}{2k_m})_{m\geq 1}$ tend vers zéro quand n tend vers l'infini et la série a un majorant fixe: la série $(\frac{(g_{1m}^1)^2}{2k_m})_{m\geq 1}$. Rappelons que $\sum_{m=1}^{\infty} \frac{(g_{1m}^1)^2}{2k_m} = \|e_1^1\|^2_{L^2(]0,L_y[)}$. On peut donc passer à la limite sous le signe somme, $\|e_1^{2n+1}\|^2_{L_2(\Omega_1)}$ tend vers zéro quand n tend vers l'infini. On a prouvé (formellement en fait car les hypothèses sur $u_{1,2}^0$ qui garantissent que $\|e_1^1\|^2_{L^2(]0,L_y[)}$ soit bornée ne sont pas données) le

Théorème 4.2.4 *Pour $\alpha > 0$, l'algorithme (4.40)-(4.41) converge. Si les sous-domaines se recouvrent, la convergence est géométrique.*

Remarque 4.2.7 *Contrairement à l'algorithme de Schwarz du paragraphe 4.2.2, on a convergence avec ou sans recouvrement des sous-domaines.*

Remarque 4.2.8 *Contrairement au cas unidimensionnel:*

- *la convergence n'est pas géométrique quand il n'y a pas de recouvrement.*
- *il n'est pas possible de choisir α pour avoir une convergence en deux itérations. En choisissant $\alpha = k_{m_0}$ pour m_0 arbitraire, on peut seulement avoir une convergence en deux itérations de la composante de la solution selon z_{m_0}*

Remarque 4.2.9 *Quand les sous-domaines se recouvrent, pour des grandes valeurs du nombre de Fourier $k_m = \frac{m\pi}{L_y}$, le facteur de convergence β_m est très petit. Rappelons que cela correspond à la composante de l'erreur selon $z_m = \sin(k_m y)$. Les composantes les plus oscillantes de l'erreur le long des interfaces $x = L_1$ et $x = l_2$ sont ainsi très rapidement atténuées par l'algorithme.*
Quand les sous-domaines ne se recouvrent pas, la convergence des composantes de l'erreur sur les modes très oscillants le long des interfaces est très lente ($\beta_m \to 1$, pour $m \to \infty$).

Remarque 4.2.10 *Les conclusions précédent valables pour un domaine borné dans la direction x.*

4.3 Aspect numérique

4.3.1 Approximations par différences finies

Nous considérons l'équation

$$\begin{array}{l} -\Delta u = f, \quad \text{dans }]0, L_x[\times]0, L_y[\\ u(x,0) = u(x, L_y) = 0 \quad \forall x \in]0, L_x[\end{array} \tag{4.44}$$

avec les conditions aux bords

$$u(0,y) = g_g(y) \text{ et } (\frac{\partial u}{\partial x} + \alpha u)(L,y) = g_d(y) \quad \forall y \in]0, L_y[, \tag{4.45}$$

où α est une constante positive ou nulle et g_g, g_d sont des fonctions scalaires. Nous introduisons des schémas d'approximation par différence finie. Nous cherchons à calculer une approximation de la solution aux points
$(x_i, y_j)_{1\leq i\leq N_x,\, 1\leq j\leq N_y}$ définis par

$$x_i = (i-1)\Delta x \text{ et } y_j = (j-1)\Delta y$$

où N_x et N_y sont les nombres de points de discrétisation dans les directions x et y et $\Delta x = L_x/(N_x - 1), \Delta y = L_y/(N_y - 1)$ désignent les pas de discrétisation. On note u_{ij} l'approximation numérique de u au point (x_i, y_j). Pour approcher (4.44), on utilise un schéma centré

$$-\frac{u_{i+1\,j} - 2u_{i\,j} + u_{i-1\,j}}{(\Delta x)^2} - \frac{u_{i\,j+1} - 2u_{i\,j} + u_{i\,j-1}}{(\Delta y)^2} = f(x_i, y_j). \tag{4.46}$$

Les conditions de Dirichlet dans (4.44) se traduisent par

$$u_{i\,1} = u_{i\,N_y} = 0, \quad 1 \leq i \leq N_x. \tag{4.47}$$

La condition de Dirichlet dans (4.45) est approchée par

$$u_{1j} = g_g(y_j), \quad 2 \leq j \leq N_y - 1. \tag{4.48}$$

On ne considère pas les valeurs $u_{1\,1}$ et $u_{1\,N_y}$ aux extrémités de $\{0\}\times]0, L_y[$ car on y a déjà écrit (cf. 4.47) une condition de Dirichlet homogène.
La condition de Robin dans (4.45) peut être approchée de plusieurs façons. Nous en mentionnons deux.

Possibilité n^{0}1.

$$\frac{u_{N_x\, j} - u_{N_x - 1\, j}}{\Delta x} + \alpha u_{N_x\, j} = g_d(y_j). \tag{4.49}$$

Cette formule correspond à une approximation de la dérivée première en x à l'aide d'un développement de Taylor à l'ordre 1:

$$u(x_{N_x-1}, y_j) = u(x_{N_x}, y_j) - \Delta x \frac{\partial u}{\partial x}(x_{N_x}, y_j) + O(\Delta x^2). \tag{4.50}$$

Possibilité n^{0}2.

$$\frac{u_{N_x\, j} - u_{N_x - 1\, j}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_x}, y_j) - \frac{\Delta x}{2} \frac{u_{N_x\, j-1} - 2u_{N_x\, j} + u_{N_x\, j+1}}{(\Delta y)^2} + \alpha u_{N_x\, j} = g_d(y_j). \tag{4.51}$$

Cette formule correspond à une approximation de la dérivée première en x à l'aide d'un développement de Taylor à l'ordre 2:

$$u(x_{N_x-1}, y_j) = u(x_{N_x}, y_j) - \Delta x \frac{\partial u}{\partial x}(x_{N_x}, y_j) + \frac{1}{2}(\Delta x)^2 \frac{\partial^2 u}{\partial x^2}(x_{N_x}, y_j) + O(\Delta x^3). \tag{4.52}$$

dans lequel on a approché $\dfrac{\partial^2 u}{\partial x^2}(x_{N_x}, y_j)$ à l'aide de l'équation (4.46) en $(i,j) = (N_x, j)$.

La possibilité (4.51) est plus précise que la possibilité (4.49) puisqu'elle est en $O(\Delta x^2)$ alors que (4.49) est en $O(\Delta x)$.

Attention! pour estimer l'ordre on divise (4.50) et (4.52) par Δx.

4.4 Méthodes de décomposition de domaine - Cas discret

Dans ce paragraphe, nous donnons un équivalent discret des méthodes de décomposition de domaine du cas continu traité dans le paragraphe 4.2. Nous nous plaçons sur une géométrie simple, infinie dans la direction x: $\mathbb{R} \times [0, L_y]$. Celle-ci n'est pas réaliste d'un point de vue pratique mais elle simplifie l'étude des méthodes de décomposition de domaine. Les pas de maillage en x et en y sont Δx et en Δy et on pose $x_i = (i-1)\Delta x$, $i \in \mathbb{Z}$ et $y_j = (j-1)\Delta y$, $1 \leq j \leq N_y - 1$.
Nous considérons le système linéaire issu de la discrétisation du problème P (Eqs. (4.22)-(4.23)):

$$-\frac{u_{i-1\, j} - 2u_{i\, j} + u_{i+1\, j}}{(\Delta x)^2} - \frac{u_{i\, j-1} - 2u_{i\, j} + u_{i\, j+1}}{(\Delta y)^2} = f(x_i, y_j),\ i \in \mathbb{Z},\ 2 \leq j \leq N_y - 1 \tag{4.53}$$

et

$$u_{i\,1} = u_{i\, N_y} = 0, \quad i \in \mathbb{Z}. \tag{4.54}$$

où $N - y$ est un entier et $(N_y - 1)\Delta y = L_y$.
Nous désignerons par $\mathbf{P}_h$ le problème (4.53)-(4.54).
L'ensemble des indices $\mathcal{N} = \{(i,j)/\ i \in \mathbb{Z},\ 1 \leq j \leq N_y\}$ est décomposé en deux sous-ensembles d'indices $\mathcal{N}_1 = \{(i,j)/\ i \leq N_{x1},\ 1 \leq j \leq N_y\}$ et $\mathcal{N}_2 = \{(i,j)/\ N_{x2} \leq i,\ 1 \leq j \leq N_y\}$ où N_{x1} et N_{x2} sont des entiers satisfaisants $N_{x2} \leq N_{x1}$. Cela correspond à une décomposition de la bande infinie $\mathbb{R} \times [0, L_y]$ en deux sous-domaines $\Omega_1 = [-\infty, L_1] \times [0, L_y]$ et $\Omega_2 = [l_2, \infty] \times [0, L_y]$ où $L_1 = N_{x1}\Delta x$ et $l_2 = N_{x2}\Delta x$.

Si $N_{x2} < N_{x1}$, on dira, par analogie avec le cas continu, que les sous-domaines se recouvrent.
Nous allons réécrire P_h de deux manières différentes sous la forme de deux problèmes couplés posés sur les sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$. Nous considérons tout d'abord le couplage par des conditions de Dirichlet puis par des conditions de Robin discrétisées. Puis, nous étudierons une méthode itérative de résolution de ces systèmes couplés adaptée aux architectures parallèles. Pour cela, nous suivrons un plan parallèle celui du § 4.2 où le cas continu est traité.

On désigne par $U_1 = (u_{ij})_{(i,j)\in\mathcal{N}_1}$ la restriction de U au sous-ensemble d'indices $\mathcal{N}_1$ et par $U_2 = (u_{ij})_{(i,j)\in\mathcal{N}_2}$ sa restriction à $\mathcal{N}_2$.
On notera

$$A_1\, U_1 = F_1 \tag{4.55}$$

le système linéaire formé par:

- l'équation de Laplace discrète (4.53) pour tous les couples d'indices (i, j) à l'intérieur de $\mathcal{N}_1$ i.e. $i < N_{x1}$ et $2 \leq j \leq N_y - 1$
- l'équation de Dirichlet discrète (4.54) pour $i \leq N_{x1}$.

De manière similaire, on notera

$$A_2\, U_2 = F_2 \tag{4.56}$$

le système linéaire formé par:

- l'équation de Laplace discrète (4.53) pour tous les couples d'indices (i, j) à l'intérieur de $\mathcal{N}_2$ i.e. $N_{x2} < i$ et $2 \leq j \leq N_y - 1$
- l'équation de Dirichlet discrète (4.54) pour $N_{x2} \leq i$.

Le système linéaire (4.55) est sous-déterminé puisqu'il ne contient pas d'équations pour les indices (N_{x1}, j), $2 \leq j \leq N_y - 1$ qui correspondent à la frontière du sous-domaine Ω_1. De même, le système linéaire (4.56) est sous-déterminé puisqu'il ne contient pas d'équations pour les indices (N_{x2}, j), $2 \leq j \leq N_y - 1$ qui correspondent à la frontière du sous-domaine Ω_2. Dans les paragraphes qui suivent, nous ajoutons des conditions de raccord aux interfaces en $i = N_{x1}$ et $i = N_{x2}$ pour que le couple (U_1, U_2) soit déterminé de manière unique et soit égal aux restrictions de U aux sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$. Pour chaque jeu de conditions d'interfaces, on propose une méthode itérative de résolution adaptée aux architectures parallèles.

4.4.1 Problème discret sans second membre posé dans une bande semi-infinie

L'étude des méthodes de décomposition de domaine faite dans les paragraphes 3.4, 3.2 et 4.2, utilise la description des solutions de l'équation de Laplace sans second membre. Le but de ce paragraphe est de décrire l'ensemble des solutions discrètes des systèmes linéaires sous-déterminés

$$A_1 W_1 = 0 \tag{4.57}$$

et

$$A_2 W_2 = 0 \tag{4.58}$$

où A_1 et A_2 sont définis par (4.55) et (4.56). Dans le cas continu (paragraphe 4.2), nous avons utilisé une décomposition en série de Fourier dans la direction y (voir les équations (4.4), (4.5), ...). Dans le cas discret qui nous intéresse, on introduit la suite de vecteurs $Z_m = \sin(m\frac{\pi}{L_y}\Delta y(j-1))_{1\leq j\leq N_y}$ pour $1 \leq m \leq N_y - 2$. On a le

Lemme 4.4.1

$$\begin{aligned} & Z_{m,1} = Z_{m,N_y} = 0 \\ & -\frac{Z_{m,j+1} - 2Z_{m,j} + Z_{m,j-1}}{(\Delta y)^2} = \mu_m Z_{m,j}, \quad 2 \leq j \leq N_y - 1 \end{aligned} \tag{4.59}$$

où

$$\mu_m = \frac{4}{(\Delta y)^2}\sin(m\frac{\pi\Delta y}{2\,L_y})^2. \tag{4.60}$$

En d'autres termes, le vecteur Z_m est un vecteur propre du laplacien discret dans la direction y avec des conditions de Dirichlet homogènes pour la valeur propre μ_m.

On appelle $V_{N_y}^h = \{(v_j)_{1\le j\le N_y} \in \mathbb{R}^{N_y} /\ v_1 = v_{N_Y} = 0\}$. Les vecteurs $(Z_m)_{2\le m\le N_y-1}$ forment une base de $V_{N_y}^h$.

Remarque 4.4.1 *On a un rapport direct entre le vecteur Z_m ci-dessus et les fonctions z_m (4.4) puisque $z_m(x_j) = Z_{m,\,j}$ et quand Δy tend vers zéro, μ_m tend vers $(\dfrac{m\pi}{L_y})^2 = -\dfrac{z_m''}{z_m}$.*

Pour $m\Delta y$ proche de L_y (ou de manière équivalente, m proche de $N_y - 1$), on a $\mu_m \simeq \dfrac{4}{(\Delta y)^2}$ alors que $(\dfrac{m\pi}{L_y})^2 \simeq \dfrac{\pi^2}{(\Delta y)^2}$. Les limites sont différentes mais assez proches puisque $\pi^2/4 \simeq 2,47$.

Les vecteurs $(Z_m)_{2\le m\le N_y-1}$ formant une base de $V_{N_y}^h$, il est possible de chercher W_2 sous la forme

$$W_{2,ij} = \sum_{m=2}^{N_y-1} w_{2,\,i}^m\, Z_{m,j}, \quad i \ge N_{x2}, 1 \le j \le N_y. \tag{4.61}$$

On cherche à ce que l'équation (4.58) soit vérifiée.
La condition de Dirichlet discrète (4.54) pour $N_{x2} \le i$ est vérifiée puisque $Z_{m\,1} = Z_{m\,N_y} = 0$ pour tout m.
L'équation de Laplace discrète (4.53) s'écrit

$$-\sum_{m=2}^{N_y-1} \frac{w_{2,\,i+1}^m - 2w_{2,\,i}^m + w_{2,\,i-1}^m}{(\Delta x)^2}\, Z_{m,j} - \sum_{m=2}^{N_y-1} w_{2,\,i}^m \frac{Z_{m,\,j+1} - 2Z_{m,\,j} + Z_{m,\,j-1}}{(\Delta y)^2} = 0,$$

soit en tenant compte du lemme 4.4.1

$$-\sum_{m=2}^{N_y-1} \frac{w_{2,\,i+1}^m - 2w_{2,\,i}^m + w_{2,\,i-1}^m}{(\Delta x)^2}\, Z_{m,j} + \sum_{m=2}^{N_y-1} w_{2,\,i}^m\ \mu_m\ Z_{m,\,j} = 0$$

et en mettant en facteur $w_{2,\,i}^m$

$$\sum_{m=2}^{N_y-1} \left(-\frac{w_{2,\,i+1}^m - 2w_{2,\,i}^m + w_{2,\,i-1}^m}{(\Delta x)^2} + \mu_m\ w_{2,\,i}^m\right) Z_{m,\,j} = 0$$

Les vecteurs $(Z_m)_{2\le m\le N_y-1}$ étant indépendants, on a

$$-\frac{w_{2,\,i+1}^m - 2w_{2,\,i}^m + w_{2,\,i-1}^m}{(\Delta x)^2} + \mu_m\ w_{2,\,i}^m = 0, \quad 2 \le m \le N_y - 1, \quad i > N_{x2} \tag{4.62}$$

On cherche des solutions particulières de cette équation aux différences finies sous la forme $w_{2,\,i}^m = r_m^i$, r_m doit satisfaire l'équation du second degré:

$$-\frac{\dfrac{1}{r_m} - 2 + r_m}{(\Delta x)^2} + \mu_m = 0 \tag{4.63}$$

dont les solutions sont

$$r_{\pm,m} = \frac{2 + (\Delta x)^2\mu_m \pm \sqrt{(2 + (\Delta x)^2\mu_m)^2 - 4}}{2}.$$

ou encore,

$$r_{\pm,m} = \frac{2 + (\Delta x)^2\mu_m \pm \sqrt{4(\Delta x)^2\mu_m + (\Delta x)^4\mu_m^2}}{2}. \quad (4.64)$$

On a $r_{+,m} > 1$ et $0 < r_{-,m} < 1$.

Remarque 4.4.2 *Par symétrie du Laplacien discret et de l'équation en r_m (4.63), on a*

$$r_{-,m} = \frac{1}{r_{+,m}}. \quad (4.65)$$

La solution générale de l'équation de récurrence (4.62) sur $w^m_{2,i}$ s'écrit sous la forme

$$w^m_{2,i} = \gamma_m r_{-,m}^{(i-N_{x2})} + \delta_m r_{+,m}^{(i-N_{x2})}.$$

Le terme $r_{+,m}^{(i-N_{x2})}$ tend vers l'infini quand i tend vers l'infini. Pour que la solution reste bornée, il faut que δ_m soit nul. Il reste $w^m_{2,i} = \gamma_m r_{-,m}^{(i-N_{x2})}$. Soit d'après (4.61), un élément du noyau de W_2 s'écrit sous la forme

$$W_{2,ij} = \sum_{m=2}^{N_y-1} \gamma_m r_{-,m}^{i-N_{x2}} Z_{m,j}, \quad i \geq N_{x2}, 1 \leq j \leq N_y \quad (4.66)$$

où $(\gamma_m)_{2\leq m\leq N_y}$ est une suite arbitraire de coefficients de Fourier. En raisonnant de la même manière pour (4.57), un élément du noyau de W_1 s'écrit sous la forme

$$W_{1,ij} = \sum_{m=2}^{N_y-1} \delta_m r_{+,m}^{i-N_{x1}} Z_{m,j}, \quad i \leq N_{x1}, 1 \leq j \leq N_y \quad (4.67)$$

où $(\delta_m)_{2\leq m\leq N_y}$ est une suite arbitraire de coefficients de Fourier. Dans ce dernier cas, c'est le terme $r_{-,m}^{i-N_{x1}}$ qui est exponentiellement croissant et doit être éliminé.

4.4.2 Reformulation avec des conditions de Dirichlet - Problème P'$_h$

Nous ajoutons aux équation (4.55)-(4.56) les conditions de raccord de type Dirichlet:

$$u_{1,N_1 j} = u_{2,N_1 j}, \quad 2 \leq j \leq N_y - 1 \quad (4.68)$$

et

$$u_{2,N_2 j} = u_{1,N_2 j}, \quad 2 \leq j \leq N_y - 1. \quad (4.69)$$

Nous désignerons par **P'**$_h$ le problème (4.55)-(4.56)-(4.68)-(4.69) Nous allons montrer que ce problème admet un couple solution unique (U_1, U_2) si et seulement si les sous-domaines se recouvrent ($N_1 > N_2$).

Existence

On construit une solution de P'$_h$ à partir de la solution U du problème P$_h$ en posant $U_1 = (u_j)_{j\in\mathcal{N}_1}$ et $U_2 = (u_j)_{j\in\mathcal{N}_2}$. Remarquons que s'il y a unicité, la solution de P'$_h$ est la restriction de U aux sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$.

Unicité

Par linéarité des équations, il suffit de considérer le cas homogène $f = 0$. D'après (4.55) et (4.67), U_1 s'écrit sous la forme

$$U_{1,ij} = \sum_{m=2}^{N_y-1} \delta_{1,m} r_{+,m}^{i-N_{x1}} Z_{m,j}, \quad i \leq N_{x1}, 1 \leq j \leq N_y$$

où $(\delta_{1,m})_{2\leq m\leq N_y}$ est une suite arbitraire de coefficients de Fourier. De même, d'après (4.56) et (4.66), U_2 s'écrit sous la forme

$$U_{2,ij} = \sum_{m=2}^{N_y-1} \delta_{2,m} r_{-,m}^{i-N_{x2}} Z_{m,j}, \quad i \geq N_{x2}, 1 \leq j \leq N_y$$

où $(\delta_{2,m})_{2\leq m\leq N_y}$ est une suite arbitraire de coefficients de Fourier.
La condition de raccord (4.68) devient:

$$\sum_{m=2}^{N_y-1} \delta_{1,m} Z_{m,j} = \sum_{m=2}^{N_y-1} \delta_{2,m} r_{-,m}^{N_{x1}-N_{x2}} Z_{m,j}, \quad 1 \leq j \leq N_y$$

soit encore

$$\sum_{m=2}^{N_y-1} (\delta_{1,m} - \delta_{2,m} r_{-,m}^{N_{x1}-N_{x2}}) Z_{m,j} = 0, \quad 1 \leq j \leq N_y$$

d'où

$$\delta_{1,m} - \delta_{2,m} r_{-,m}^{N_{x1}-N_{x2}} = 0, \quad 2 \leq m \leq N_y - 1. \tag{4.70}$$

De même, la condition de raccord (4.69) s'écrit

$$\delta_{2,m} - \delta_{1,m} r_{+,m}^{N_{x2}-N_{x1}} = 0, \quad 2 \leq m \leq N_y - 1. \tag{4.71}$$

Pour chaque m, le système linéaire en $(\delta_{1,m}, \delta_{2,m})$ (4.70)-(4.71) a un déterminant qui vaut $1 - \left(\frac{r_{-,m}}{r_{+,m}}\right)^{N_{x1}-N_{x2}}$.
Il est non nul ssi les sous-domaines se recouvrent, $N_{x2} < N_{x1}$. Alors, on a $\delta_{1,m} = \delta_{2,m} = 0$ pour tout m et donc $U_1 = U_2 = 0$. Ceci prouve l'unicité.
Dans le cas où les sous-domaines sont adjacents ($N_{x2} = N_{x1}$), le déterminant de (4.70)-(4.71) s'annule et le système linéaire (4.30)-(4.31) admet une infinité de solutions $\delta_{1,m} = \delta_{2,m}, \delta_{2,m} = \delta_{2,m}$. Il en est alors de même du problème P'$_h$ qui admet une infinité de solutions.

Lemme 4.4.2 *Le problème P'$_h$ (Eqs. (4.55)-(4.56)-(4.68)-(4.69) admet une unique solution ssi les sous-domaines se recouvrent. Dans ce cas, la solution (U_1, U_2) de P'$_h$ est composée des restrictions, aux sous-domaines, de la solution U du problème P_h (i.e. $U_1 = U_{|\mathcal{N}_1}$ et $U_2 = U_{|\mathcal{N}_2}$).*

4.4.3 Algorithme parallèle de résolution du problème P'$_h$ (Algorithme de Schwarz discret)

Nous considérons une méthode itérative de résolution du problème P'$_h$. On part de U_1^0 et U_2^0 quelconques et on pose

$$\begin{cases} A_1 U_1^{n+1} = F_1, \\ u_{1,N_1 j}^{n+1} = u_{2,N_1 j}^n, \quad 2 \leq j \leq N_y - 1 \end{cases} \tag{4.72}$$

et

$$\begin{cases} A_2 U_2^{n+1} = F_2, \\ u_{2,N_2 j}^{n+1} = u_{1,N_2 j}^n, \quad 2 \leq j \leq N_y - 1 \end{cases} \tag{4.73}$$

où les matrices A_1 et A_2 sont définies par (4.55) et (4.56).

Remarque 4.4.3 *Une solution stationnaire de l'algorithme est solution du problème P'$_h$. Ceci prouve que si l'algorithme converge (voir plus loin la discussion suivant le chevauchement des sous-domaines), il converge vers l'unique solution de P'$_h$ qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution U du problème P_h. L'algorithme de Schwarz (4.72)-(4.73) sera donc une méthode itérative de résolution du problème P_h.*

Pour étudier la convergence, on pose $E_1^n = (e_{1,ij}^n)_{(i,j)\in\mathcal{N}_1} = U_1^n - U_1$ et $E_2^n = (e_{2,ij}^n)_{(i,j)\in\mathcal{N}_2} = U_2^n - U_2$ où (U_1, U_2) est la solution du problème P'$_h$. Par linéarité des équations, l'erreur vérifie

$$\begin{cases} A_1\, E_1^{n+1} = 0, \\ e_{1,N_1 j}^{n+1} = e_{2,N_1 j}^n, \quad 2 \le j \le N_y - 1 \end{cases} \tag{4.74}$$

et

$$\begin{cases} A_2\, E_2^{n+1} = 0, \\ e_{2,N_2 j}^{n+1} = e_{1,N_2 j}^n, \quad 2 \le j \le N_y - 1. \end{cases} \tag{4.75}$$

Les vecteurs E_1^n et E_2^n sont dans le noyau de A_1 et A_2 respectivement. D'après le paragraphe 4.4.1 (eqs. (4.67) et (4.66)), on peut écrire pour $n \ge 1$:

$$e_{1,ij}^n = \sum_{m=2}^{N_y-1} \delta_{1,m}^n r_{+,m}^{i-N_{x1}}\, Z_{m,j}, \quad i \le N_{x1}, 1 \le j \le N_y$$

et

$$e_{2,ij}^n = \sum_{m=2}^{N_y-1} \delta_{2,m}^n r_{-,m}^{i-N_{x2}}\, Z_{m,j}, \quad i \ge N_{x2}, 1 \le j \le N_y.$$

Les conditions de Dirichlet en $i = N_{x1}$ et $i = N_{x2}$ donnent:

$$\delta_{1,\,m}^{n+1} = \delta_{2,\,m}^n r_{-,m}^{N_{x1}-N_{x2}}, \quad 2 \le m \le N_y - 1$$

et

$$\delta_{2,\,m}^{n+1} = \delta_{1,\,m}^n\, r_{+,m}^{N_{x2}-N_{x1}}, \quad 2 \le m \le N_y - 1.$$

En utilisant successivement les deux relations précédentes, on a pour tout m et $n \ge 2$:

$$\delta_{1,\,m}^{n+1} = \delta_{2,\,m}^n r_{-,m}^{N_{x1}-N_{x2}} = \left(\frac{r_{-,\,m}}{r_{+,\,m}}\right)^{N_{x1}-N_{x2}} \delta_{1,\,m}^{n-1} \equiv \beta_m\ \delta_{1,\,m}^{n-1}.$$

La relation est aussi valable pour $\delta_{2,\,m}^{n+1}$. Comme $N_{x2} \le N_{x1}$, le terme β_m est toujours positif et on a: $\beta_m < 1 \iff N_{x2} < N_{x1}$. Pour $N_{x2} = N_{x1}$, on a $\beta_m = 1$ et l'algorithme stagne. On a prouvé le

Théorème 4.4.3 *L'algorithme de Schwarz (4.72)-(4.73) converge si et seulement si les sous-domaines se recouvrent ($N_{x2} < N_{x1}$).*

Remarque 4.4.4 *L'absence de convergence dans le cas sans recouvrement est en accord avec le fait que le problème P'$_h$ est alors mal posé.*

Remarque 4.4.5 *D'après (4.64), on a*

$$\frac{r_{-,\,m}}{r_{+,\,m}} = \frac{1 - \sqrt{1 - \frac{4}{(2+(\Delta x)^2\,\mu_m)^2}}}{1 + \sqrt{1 - \frac{4}{(2+(\Delta x)^2\,\mu_m)^2}}}$$

Plus m est grand, plus μ_m (Eq. (4.60)) est grand. Ainsi, plus m est grand, plus le facteur de convergence β_m est petit. Rappelons que cela correspond à la composante de l'erreur selon $Z_m = (\sin(k_m y_j))_{1\le j\le N_y}$. Les composantes les plus oscillantes de l'erreur le long des interfaces $x = L_1$ et $x = l_2$ sont ainsi très rapidement atténuées par l'algorithme.

Remarque 4.4.6 *Les conclusions précédent valables pour un domaine borné dans la direction x.*

4.4.4 Reformulation avec des conditions de Robin - Problème P"$_h$

Le but de ce paragraphe est de coupler les systèmes linéaires (4.55)-(4.56) par des conditions de Robin (4.36)-(4.37) discrétisées sur les segments $i = N_{x1}$ et $i = N_{x2}$ $(2 \le j \le N_y - 1)$ par (cf. (4.51)):

$$\begin{aligned}&\frac{u_{1,N_{x1}\,j} - u_{1,N_{x1}-1\,j}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_{x1}}, y_j) - \frac{\Delta x}{2} \frac{u_{1,N_{x1}\,j-1} - 2u_{1,N_{x1}\,j} + u_{1,N_{x1}\,j+1}}{(\Delta y)^2}\\ &+\alpha u_{1,N_{x1}\,j} =\\ &-\left(\tfrac{u_{2,N_{x1}\,j} - u_{2,N_{x1}+1\,j}}{\Delta x} - \tfrac{\Delta x}{2} f(x_{N_{x1}}, y_j) - \tfrac{\Delta x}{2} \tfrac{u_{2,N_{x1}\,j-1} - 2u_{2,N_{x1}\,j} + u_{2,N_{x1}\,j+1}}{(\Delta y)^2}\right)\\ &+\alpha u_{2,N_{x1}\,j}\end{aligned} \tag{4.76}$$

et (attention au changement de direction de la dérivée normale: $n_2 = -n_1$)

$$\begin{aligned}&\frac{u_{2,N_{x2}\,j} - u_{2,N_{x2}+1\,j}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_{x2}}, y_j) - \frac{\Delta x}{2} \frac{u_{2,N_{x2}\,j-1} - 2u_{2,N_{x2}\,j} + u_{2,N_{x2}\,j+1}}{(\Delta y)^2}\\ &+\alpha u_{2,N_{x2}\,j} =\\ &-\left(\tfrac{u_{1,N_{x2}\,j} - u_{1,N_{x1}-1\,j}}{\Delta x} - \tfrac{\Delta x}{2} f(x_{N_{x2}}, y_j) - \tfrac{\Delta x}{2} \tfrac{u_{1,N_{x2}\,j-1} - 2u_{1,N_{x2}\,j} + u_{1,N_{x2}\,j+1}}{(\Delta y)^2}\right)\\ &+\alpha u_{1,N_{x2}\,j}\end{aligned} \tag{4.77}$$

où α est un nombre strictement positif. Nous désignerons encore par **P"**$_h$ le problème constitué par les équations (4.55)-(4.56)-(4.76)-(4.77). Nous allons montrer que l'unique solution du problème P"$_h$ est la restriction aux sous-ensembles d'indices $\mathcal{N}_1$ et $\mathcal{N}_2$ de la solution du problème P$_h$ (Eqs. (4.53)-(4.54)) posé sur l'ensemble des inconnues. Ce résultat est valable avec ou sans recouvrement des sous-domaines.

Remarque (Importante) 4.4.1 *Les discrétisations des dérivées normales dans (4.76)-(4.77) ne font pas appel aux même points suivant que l'on soit dans le sous-domaine 1 ou dans le sous-domaine 2. On pourrait s'inquiéter des problèmes de consistance avec le problème P_h. Comme le montre le paragraphe* **Existence** *ci-dessous, la dépendance de (4.76) et (4.77) par rapport au second membre f assure l'équivalence des formulations.*

Existence

On construit une solution de P"$_h$ à partir de la solution U de la solution du problème P$_h$ en posant $U_1 = (u_j)_{j\in\mathcal{N}_1}$ et $U_2 = (u_j)_{j\in\mathcal{N}_2}$. Le couple (U_1, U_2) ainsi formé vérifie évidemment les équations (4.55) et (4.56). Vérifions que l'équation (4.76) est satisfaite. Comme $u_{1,i\,N_{x1}} = u_{2,i\,N_{x1}}$ ceci équivaut à montrer que

$$\begin{aligned}&\frac{u_{N_{x1}\,j} - u_{N_{x1}-1\,j}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_{x1}}, y_j) - \frac{\Delta x}{2} \frac{u_{N_{x1}\,j-1} - 2u_{N_{x1}\,j} + u_{N_{x1}\,j+1}}{(\Delta y)^2}\\ &= -\left(\frac{u_{N_{x1}\,j} - u_{N_{x1}+1\,j}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_{x1}}, y_j) - \frac{\Delta x}{2} \frac{u_{N_{x1}\,j-1} - 2u_{N_{x1}\,j} + u_{N_{x1}\,j+1}}{(\Delta y)^2}\right).\end{aligned}$$

En divisant par Δx et en regroupant les termes, ceci équivaut à

$$-\frac{u_{N_{x1}+1\,j} - 2u_{N_{x1}\,j} + u_{N_{x1}-1\,j}}{(\Delta x)^2} - \frac{u_{N_{x1}\,j+1} - 2u_{N_{x1}\,j} + u_{N_{x1}\,j-1}}{(\Delta y)^2} = f(x_{N_{x1}}, y_j).$$

Il s'agit du schéma de discrétisation du Laplacien écrit au point (x_{N_1}, y_j). On montre de même que l'équation (4.77) est satisfaite.

Unicité

Par linéarité des équations, il suffit de considérer les équations homogènes $f = 0$. D'après (4.55) et (4.67), U_1 s'écrit sous la forme

$$U_{1,ij} = \sum_{m=2}^{N_y-1} \delta_{1,m} r_{+,m}^{i-N_{x1}} Z_{m,j}, \quad i \le N_{x1}, 1 \le j \le N_y$$

où $(\delta_{1,m})_{2\leq m\leq N_y}$ est une suite arbitraire de coefficients de Fourier. De même, d'après (4.56) et (4.66), U_2 s'écrit sous la forme

$$U_{2,ij}=\sum_{m=2}^{N_y-1}\delta_{2,m}r_{-,m}^{i-N_{x2}}Z_{m,j},\quad i\geq N_{x2},1\leq j\leq N_y$$

où $(\delta_{2,m})_{2\leq m\leq N_y}$ est une suite arbitraire de coefficients de Fourier.
La condition de raccord (4.76) s'écrit en utilisant (4.59) et en groupant les termes:

$$\sum_{m=2}^{N_y-1}\left((\frac{1-r_{+,m}^{-1}}{\Delta x}+\frac{\Delta x}{2}\mu_m)+\alpha\right)\delta_{1,m}Z_{m,j}$$
$$=\sum_{m=2}^{N_y-1}\left(-(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m)+\alpha\right)r_{-,m}^{N_{x1}-N_{x2}}\delta_{2,m}Z_{m,j},$$

soit encore pour tout m:

$$(\frac{1-r_{+,m}^{-1}}{\Delta x}+\frac{\Delta x}{2}\mu_m+\alpha)\;\delta_{1,m}+(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m-\alpha)r_{-,m}^{N_{x1}-N_{x2}}\;\delta_{2,m}=0. \quad (4.78)$$

De même, la condition de raccord (4.77) s'écrit:

$$(\frac{1-r_{+,m}^{-1}}{\Delta x}+\frac{\Delta x}{2}\mu_m-\alpha)r_{+,m}^{N_{x2}-N_{x1}}\;\delta_{1,m}+(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m+\alpha)\;\delta_{2,m}=0. \quad (4.79)$$

Pour chaque m, le système linéaire en $(\delta_{1,m},\delta_{2,m})$ (4.78)-(4.79) a un déterminant det qui vaut

$$det=(\frac{1-r_{+,m}^{-1}}{\Delta x}+\frac{\Delta x}{2}\mu_m+\alpha)(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m+\alpha)$$
$$-(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m-\alpha)(\frac{1-r_{+,m}^{-1}}{\Delta x}+\frac{\Delta x}{2}\mu_m-\alpha)\left(\frac{r_{-,m}}{r_{+,m}}\right)^{N_{x1}-N_{x2}}$$

soit encore en utilisant (4.65),

$$det=(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m+\alpha)^2-(\frac{1-r_{-,m}}{\Delta x}+\frac{\Delta x}{2}\mu_m-\alpha)^2\left(\frac{r_{-,m}}{r_{+,m}}\right)^{N_{x1}-N_{x2}}$$

Le déterminant est strictement positif. Le système (4.78)-(4.79) admet comme unique solution la solution nulle. Il en est alors de même du problème P"$_h$. On a ainsi montré le

Lemme 4.4.4 *Le problème $P"_h$ (Eqs. (4.55)-(4.56)-(4.68)-(4.69)) admet une unique solution. La solution (U_1,U_2) de $P"_h$ est composée des restrictions, aux sous-domaines, de la solution U du problème P_h (i.e. $U_1=U_{|\mathcal{N}_1}$ et $U_2=U_{|\mathcal{N}_2}$).*

Remarque 4.4.7 *Il est important de noter que le problème $P"_h$, contrairement au problème P'_h, admet une solution unique même quand les sous-domaines ne se recouvrent pas.*

4.4.5 Algorithme parallèle de résolution du problème P"$_h$

Nous considérons une méthode itérative de résolution du problème P"$_h$. On part de U_1^0 et U_2^0 quelconques et on pose

$$\left\{\begin{array}{l} A_1\, U_1^{n+1} = F_1, \\ \dfrac{u_{1,N_{x1}\,j}^{n+1} - u_{1,N_{x1}-1\,j}^{n+1}}{\Delta x} - \dfrac{\Delta x}{2} f(x_{N_{x1}}, y_j) - \dfrac{\Delta x}{2} \dfrac{u_{1,N_{x1}\,j-1}^{n+1} - 2u_{1,N_{x1}\,j}^{n+1} + u_{1,N_{x1}\,j+1}^{n+1}}{(\Delta y)^2} \\ +\alpha u_{1,N_{x1}\,j}^{n+1} = \\ -(\frac{u_{2,N_{x1}\,j}^{n} - u_{2,N_{x1}+1\,j}^{n}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_{x1}}, y_j) - \frac{\Delta x}{2} \frac{u_{2,N_{x1}\,j-1}^{n} - 2u_{2,N_{x1}\,j}^{n} + u_{2,N_{x1}\,j+1}^{n}}{(\Delta y)^2}) \\ +\alpha u_{2,N_{x1}\,j}^{n} \end{array}\right. \tag{4.80}$$

et

$$\left\{\begin{array}{l} A_2\, U_2^{n+1} = F_2, \\ \dfrac{u_{2,N_{x2}\,j}^{n+1} - u_{2,N_{x2}+1\,j}^{n+1}}{\Delta x} - \dfrac{\Delta x}{2} f(x_{N_{x2}}, y_j) - \dfrac{\Delta x}{2} \dfrac{u_{2,N_{x2}\,j-1}^{n+1} - 2u_{2,N_{x2}\,j}^{n+1} + u_{2,N_{x2}\,j+1}^{n+1}}{(\Delta y)^2} \\ +\alpha u_{2,N_{x2}\,j}^{n+1} = \\ -(\frac{u_{1,N_{x2}\,j}^{n} - u_{1,N_{x1}-1\,j}^{n}}{\Delta x} - \frac{\Delta x}{2} f(x_{N_{x2}}, y_j) - \frac{\Delta x}{2} \frac{u_{1,N_{x2}\,j-1}^{n} - 2u_{1,N_{x2}\,j}^{n} + u_{1,N_{x2}\,j+1}^{n}}{(\Delta y)^2}) \\ +\alpha u_{1,N_{x2}\,j}^{n} \end{array}\right. \tag{4.81}$$

où les matrices A_1 et A_2 sont définies par (4.55) et (4.56).

Remarque 4.4.8 *Une solution stationnaire de l'algorithme est solution du problème P"$_h$. Ceci prouve que si l'algorithme converge, il converge vers l'unique solution de P"$_h$ qui est la restriction aux sous-domaines Ω_1 et Ω_2 de la solution U du problème P_h. L'algorithme itératif (4.80)-(4.81) sera donc une méthode itérative de résolution du problème P_h.*

Pour étudier la convergence, on pose $E_1^n = (e_{1,ij}^n)_{(i,j)\in\mathcal{N}_1} = U_1^n - U_1$ et $E_2^n = (e_{2,ij}^n)_{(i,j)\in\mathcal{N}_2} = U_2^n - U_2$ où (U_1, U_2) est la solution du problème P"$_h$. Par linéarité des équations, l'erreur vérifie

$$\left\{\begin{array}{l} A_1\, E_1^{n+1} = 0, \\ \dfrac{e_{1,N_{x1}\,j}^{n+1} - e_{1,N_{x1}-1\,j}^{n+1}}{\Delta x} - \dfrac{\Delta x}{2} \dfrac{e_{1,N_{x1}\,j-1}^{n+1} - 2e_{1,N_{x1}\,j}^{n+1} + e_{1,N_{x1}\,j+1}^{n+1}}{(\Delta y)^2} \\ +\alpha e_{1,N_{x1}\,j}^{n+1} \\ = -(\dfrac{e_{2,N_{x1}\,j}^{n} - e_{2,N_{x1}+1\,j}^{n}}{\Delta x} - \dfrac{\Delta x}{2} \dfrac{e_{2,N_{x1}\,j-1}^{n} - 2e_{2,N_{x1}\,j}^{n} + e_{2,N_{x1}\,j+1}^{n}}{(\Delta y)^2}) \\ +\alpha e_{2,N_{x1}\,j}^{n} \end{array}\right. \tag{4.82}$$

et

$$\left\{\begin{array}{l} A_2\, E_2^{n+1} = 0, \\ \dfrac{e_{2,N_{x2}\,j}^{n+1} - e_{2,N_{x2}+1\,j}^{n+1}}{\Delta x} - \dfrac{\Delta x}{2} \dfrac{e_{2,N_{x2}\,j-1}^{n+1} - 2e_{2,N_{x2}\,j}^{n+1} + e_{2,N_{x2}\,j+1}^{n+1}}{(\Delta y)^2} \\ +\alpha e_{2,N_{x2}\,j}^{n+1} \\ = -(\dfrac{e_{1,N_{x2}\,j}^{n} - e_{1,N_{x1}-1\,j}^{n}}{\Delta x} - \dfrac{\Delta x}{2} \dfrac{e_{1,N_{x2}\,j-1}^{n} - 2e_{1,N_{x2}\,j}^{n} + e_{1,N_{x2}\,j+1}^{n}}{(\Delta y)^2}) \\ +\alpha e_{1,N_{x2}\,j}^{n} \end{array}\right. \tag{4.83}$$

Les vecteurs E_1^n et E_2^n sont dans le noyau de A_1 et A_2 respectivement. D'après le paragraphe 4.4.1 (eqs. (4.67) et (4.66)), on peut écrire pour $n \geq 1$:

$$e_{1,ij}^n = \sum_{m=2}^{N_y-1} \delta_{1,m}^n r_{+,m}^{i-N_{x1}} Z_{m,j}, \quad i \leq N_{x1}, 1 \leq j \leq N_y$$

et

$$e^n_{2,ij} = \sum_{m=2}^{N_y-1} \delta^n_{2,m} r_{-,m}^{i-N_{x2}} Z_{m,j}, \quad i \geq N_{x2}, 1 \leq j \leq N_y.$$

La condition de Robin en $i = N_{x1}$ s'écrit

$$\sum_{m=2}^{N_y-1} \left((\frac{1-r_{+,m}^{-1}}{\Delta x} + \frac{\Delta x}{2}\mu_m) + \alpha \right) \delta^{n+1}_{1,m} Z_{m,j}$$
$$= \sum_{m=2}^{N_y-1} \left(-(\frac{1-r_{-,m}}{\Delta x} + \frac{\Delta x}{2}\mu_m) + \alpha \right) r_{-,m}^{N_{x1}-N_{x2}} \delta^n_{2,m} Z_{m,j},$$

soit encore pour tout m (et en utilisant (4.65)):

$$(\frac{1-r_{-,m}}{\Delta x} + \frac{\Delta x}{2}\mu_m + \alpha)\ \delta^{n+1}_{1,m} = -(\frac{1-r_{-,m}}{\Delta x} + \frac{\Delta x}{2}\mu_m - \alpha) r_{-,m}^{N_{x1}-N_{x2}}\ \delta^n_{2,m}.$$

De même, la condition de Robin en $i = N_{x2}$ s'écrit:

$$(\frac{1-r_{-,m}}{\Delta x} + \frac{\Delta x}{2}\mu_m + \alpha)\ \delta^{n+1}_{2,m} = -(\frac{1-r_{-,m}}{\Delta x} + \frac{\Delta x}{2}\mu_m - \alpha) r_{-,m}^{N_{x1}-N_{x2}}\ \delta^n_{1,m}.$$

En utilisant successivement les deux relations précédentes, on a pour tout m et $n \geq 2$:

$$\begin{aligned} \delta^{n+1}_{1,m} &= -\frac{\dfrac{1-r_{-,m}}{\Delta x} + \dfrac{\Delta x}{2}\mu_m - \alpha}{\dfrac{1-r_{-,m}}{\Delta x} + \dfrac{\Delta x}{2}\mu_m + \alpha} r_{-,m}^{N_{x1}-N_{x2}}\ \delta^n_{2,m} \\ &= \left(\frac{\dfrac{1-r_{-,m}}{\Delta x} + \dfrac{\Delta x}{2}\mu_m - \alpha}{\dfrac{1-r_{-,m}}{\Delta x} + \dfrac{\Delta x}{2}\mu_m + \alpha} \right)^2 r_{-,m}^{2(N_{x1}-N_{x2})}\ \delta^{n-1}_{1,m} \\ &\equiv \beta_m \delta^{n-1}_{1,m}. \end{aligned}$$

La même relation est valable pour $\delta^{n+1}_{2,m}$. Le paramètre α étant positif, on a $0 < \beta_m < 1$ pour $2 \leq m \leq N_y - 1$ avec ou sans recouvrement des sous-domaines. On a prouvé le

Théorème 4.4.5 *Pour $\alpha > 0$, l'algorithme (4.80)-(4.81) converge géométriquement.*

Remarque 4.4.9 *Dans le cas continu sans recouvrement, la convergence a lieu mais n'est pas géométrique. Ici, la convergence est géométrique même sans recouvrement car on décompose l'erreur sur un nombre fini de vecteurs Z_m . Dans le cas continu, l'erreur est décomposée sur un nombre infini de fonctions z_m.*

Remarque 4.4.10 *Contrairement à l'algorithme de Schwarz du paragraphe 4.4.2, on a convergence avec ou sans recouvrement des sous-domaines.*

Remarque 4.4.11 *Contrairement au cas unidimensionnel, il n'est pas possible de choisir α pour avoir une convergence en deux itérations. En choisissant*

$$\alpha = \frac{1-r_{-,m_0}}{\Delta x} + \frac{\Delta x}{2}\mu_{m_0}$$

pour m_0 arbitraire, on peut seulement avoir une convergence en deux itérations de la composante de la solution selon Z_{m_0}

Remarque 4.4.12 *Le recouvrement des sous-domaines favorise surtout la convergence des modes avec des grandes valeurs de* m. *Les composantes les plus oscillantes de l'erreur le long des interfaces* $x = L_1$ *et* $x = l_2$ *sont alors plus rapidement atténuées par l'algorithme.*

Remarque 4.4.13 *Les conclusions précédentes valables pour un domaine borné dans la direction* x.

Chapitre 5

PVM avancé

Ce chapitre nous offre un petit interlude, afin de développer quelques unes des fonctions avancées de PVM. Celles qui nous paraissent vraiment essentielles sont dans ce chapitre, et concernent la possibilité de recevoir des messages sans bloquer, et la possibilité d'envoyer des paramètres à la création des processus, ce qui permet en particulier d'écrire facilement des programmes parallèles récursifs.

5.1 Messages non-bloquants

On a vu la réception bloquante, Elle peut aussi être non-bloquante ou bloquante avec "timeout".

5.1.1 pvm_nrecv

Cette instruction effectue une réception non-bloquante: si un message avec le bon tag est arrivé, le place dans le tampon `bufid`. Sinon renvoie `bufid=0`:

```
int bufid;
bufid=pvm_nrecv(int tid,int msgtag);
```

Il y a un cas d'Erreur si `bufid <0`.

5.1.2 pvm_trecv

Cette instruction effectue une réception bloquante d'un message du processus `tid` avec le tag `msgtag`. Si rien n'arrive avant le délai `tmout` (champs entiers `tv_sec` et `tv_usec`) renvoie `bufid=0`:

```
int bufid;
bufid=pvm_trecv(int tid,int msgtag,struct timeval *tmout);
```

Il y a un cas d'Erreur si `bufid <0`.

5.1.3 pvm_probe

Cette instruction teste l'arrivée d'un message du processus `tid` avec le tag `msgtag`:

```
int bufid;
bufid=pvm_probe(int tid,int msgtag);
```

Il y a un cas d'Erreur si `bufid <0`.

5.1.4 pvm_bufinfo

Cette instruction donne l'état du tampon de communication `bufid`: après exécution, `bytes` contient le nombre d'octets du message éventuellement reçu, `msgtag`, son tag et `tid` l'identificateur de tâche de l'expéditeur:

```
int info;
info=pvm_bufinfo(int bufid,int *bytes,int *msgtag,int *tid);
```

Il y a un cas d'Erreur si `info <0`.

5.1.5 Exemple

On considère le programme maître suivant; il crée un esclave (`async2`), affiche qu'il est en train de calculer quelque chose, teste régulièrement si son esclave lui a envoyé un message et si tel est le cas sort de sa boucle et termine. C'est ce type de test régulier que l'on appelle "active polling".

```
#include <stdio.h>
#include <pvm3.h>
#define MSG 1

main()
{
int mtid,ctid,bufid,ok,i;
if ((mtid=pvm_mytid())<0)
{
  pvm_exit();
  exit(1); }
 pvm_spawn("async2",(char **) 0,PvmTaskDefault,(char *) 0,1,&ctid);
 fprintf(stdout,"I am spawning one child\n");
 fflush(stdout);
 ok = 0;
 i = -1;
 while (!ok)
   { i++;
   bufid = pvm_nrecv(ctid,MSG);
   if (bufid < 0) { pvm_perror("Pb with nrecv");
   pvm_exit(); exit(1); }
   if (bufid != 0) ok=1;
   else if ((i % 60000) == 0)
     { fprintf(stdout,"I am computing...\n");
     fflush(stdout);
     } }
 pvm_upkint(&i,1,1);
 fprintf(stdout,"I have received %d\n",i);
 fflush(stdout);
 pvm_exit();
 exit(1);
}
```

L'esclave "async2" est très simple:

```
#include <stdio.h>
#include <pvm3.h>
#define MSG 1
```

```
main()
{
int i=42;
sleep(4);
fprintf(stdout,"I'm sending a message to my parent\n");
fflush(stdout);
pvm_initsend(PvmDataDefault);
pvm_pkint(&i,1,1);
pvm_send(pvm_parent(),MSG);
pvm_exit();
exit(1);
}
```

L'exécution du programme donne:

```
pvm> spawn -1 -> async1
[11]
1 successful
t40021
pvm> [11:t40021] I am spawning one child
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40021] I am computing...
[11:t40022] I am sending a message to my parent
[11:t40022] EOF
[11:t40021] I have received 42
[11:t40021] EOF
[11] finished
```

5.2 Passage d'arguments et processus récursifs

5.2.1 Rappel: Transformée de Fourier Rapide

Soit un signal analogique x_a. Supposons x_a de carré sommable. La tranformée de Fourier de x_a est $\hat{x}_a$,

$$\hat{x}_a(s) = \int_{-\infty}^{\infty} x_a(t) e^{-ist} dt$$

de transformée inverse,

$$x_a(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{x}_a(s) e^{ist} ds$$

Dans le cas des signaux digitaux x avec $T = [0, N-1] \cap \mathbb{N}$, on a une décomposition semblable en sommes,

$$\hat{x}(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-\frac{2i\pi}{N} kn}$$

de transformée inverse,

$$x(n) = \sum_{k=0}^{N} \hat{x}(k) e^{\frac{2i\pi}{N} kn}$$

Les fonctions $\hat{x}$ et $\hat{x}_a$ sont les fonctions en fréquence, correspondant respectivement à x et x_a. La transformée de Fourier discrète est très utilisée dans un grand nombre de systèmes de traitement de signaux (après échantillonage et quantification). Le calcul de cette transformée peut se faire en $O(Nlog(N))$ opérations au lieu des $O(N^2)$ opérations que la définition donnée semble rendre nécessaire. Cet algorithme de calcul est connu sous le nom de tranformée de Fourier rapide (FFT: “Fast Fourier Transform”).

L'algorithme de transformée de Fourier rapide est basé sur le fait suivant.

Soit $(x(n))_{0\leq n\leq N-1}$ un signal digital avec $N = 2^m$. Alors pour $k < N/2$,

$$\begin{aligned}
\hat{x}(k) &= \frac{1}{N}\sum_{n=0}^{N-1} x(n)e^{-\frac{2i\pi}{N}kn} \\
&= \frac{1}{2}\left(\frac{1}{N/2}\sum_{n \text{ pair}} x(n)e^{-\frac{2i\pi}{N}kn} + \frac{1}{N/2}\sum_{n \text{ impair}} x(n)e^{-\frac{2i\pi}{N}kn}\right) \\
&= \frac{1}{2}\left(\frac{1}{N/2}\sum_{n=0}^{\frac{N}{2}} x(2n)e^{-\frac{2i\pi}{N}k2n} + \frac{1}{N/2}\sum_{n=0}^{\frac{N}{2}} x(2n+1)e^{-\frac{2i\pi}{N}k(2n+1)}\right) \\
&= \frac{1}{2}\left(\hat{x}_1(k) + e^{-\frac{2i\pi}{N}k}\hat{x}_2(k)\right)
\end{aligned}$$

et pour $k > N/2$,

$$\hat{x}(k) = \frac{1}{2}\left(\hat{x}_1(k) - e^{-\frac{2i\pi}{N}k}\hat{x}_2(k)\right)$$

où $x_1(n) = x(2n)$ et $x_2(n) = x(2n+1)$ sont les échantillonages du signal x aux temps pairs et impairs respectivement.

Pour calculer rapidement la transformée inverse, il suffit de remarquer que la transformée de Fourier inverse d'un signal x est N fois le conjugué de la transformée de Fourier du signal conjugué x^*.

Le calcul de la FFT s'effectue donc suivant le paradigme classique “diviser pour régner”. On a m appels récursifs à la procédure de calcul de la FFT. Il s'ensuit aisément que la complexité de l'algorithme est en moyenne de Nm opérations élémentaires.

5.2.2 Version parallèle récursive

Cet algorithme se prête également à des parallélisations efficaces. La procédure de calcul de $\hat{x}$, $fft(x)$ est définie de façon récursive. $fft(x)$ lance un processus fils qui se charge de calculer $\hat{x}_1$ et un autre qui se charge de calculer $\hat{x}_2$. Une représentation des appels récursifs est donnée dans la figure 5.1. Un programme PVM implémentant cet algorithme est donné plus bas.

“fft-startup” est à la racine de l'arbre des appels récursifs:

```
#include <stdlib.h>
#include <stdio.h>
#include "pvm3.h"

#define RESULT_TAG 1
#define MAX_LENGTH 128
```

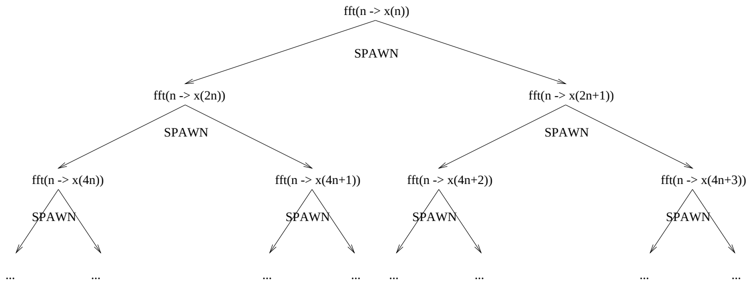


FIG. 5.1 – *Arbre de création des processus dans l'exécution de l'algorithme de FFT.*

```
main(argc, argv)
int argc;
char *argv[];
{
/* argv[i] is the unidimensional complex signal to be processed */
/* format is fft Re(x_1) Im(x_1) Re(x_2) ... Re(x_n) Im(x_n) */
int i,bufid,mtid,tid,numt;
float R[MAX_LENGTH];
if ((mtid=pvm_mytid())<0)
 {
  printf("unexpected error occurred!\n");
  exit(1);
 }
if (argc < 5)
 {
  printf("wrong number of parameters\n");
  exit(1);
 }
numt = pvm_spawn("fft",&argv[1],PvmTaskDefault,"",1,&tid);
if (numt != 1)
 {
  printf("could not spawn fft\n");
  exit(1);
 }
bufid = pvm_recv(tid, RESULT_TAG);
pvm_upkfloat(&R[0],argc-1,1);
for (i=0;i<(argc-1)/2;i++)
 {
  printf(" %g %g",R[i],R[i+(argc-1)/2]);
 }
printf("\n");
pvm_exit();
exit(0);
}
```

Le programme récursif "fft" se charge du calcul effectif:

```
#include <math.h>
#include <stdio.h>
#include "pvm3.h"

#define RESULT_TAG 1
```

```
#define SIZE 64
#define MAX_LENGTH 128
#define PI 3.14159265358

main(argc, argv)
int argc;
char *argv[];
{
/* argv[i] is the unidimensional complex signal to be processed */
/* format is fft Re(x_1) Im(x_1) Re(x_2) ... Re(x_n) Im(x_n) */
/* sends to his parent an array with RESULT_TAG equal to 1 */
  int i,numt,bufid,mtid,ptid,lstid,rstid;
  float wr,wi;
  float Rr[MAX_LENGTH],Ri[MAX_LENGTH];
  float Xr[MAX_LENGTH];
  float Xi[MAX_LENGTH];
  float T1[MAX_LENGTH],T2[MAX_LENGTH];
  char *fargv1[MAX_LENGTH+1],*fargv2[MAX_LENGTH+1];
  if ((mtid=pvm_mytid())<0)
    {
      printf("unexpected error occurred!\n");
      exit(1);
    }
  ptid=pvm_parent();
  for (i=0;i<(argc-1)/2;i++)
    {
      sscanf(argv[2*i+1],"%g",&Xr[i]);
      sscanf(argv[2*(i+1)],"%g",&Xi[i]);
      if (i%2 == 0)
{
  fargv1[i]=argv[2*i+1];
  fargv1[i+1]=argv[2*(i+1)];
}
      else
{
  fargv2[i-1]=argv[2*i+1];
  fargv2[i]=argv[2*(i+1)];
}
    }
  fargv1[(argc-1)/2]=NULL;
  fargv2[(argc-1)/2]=NULL;
  if (argc==5)
    {
      Rr[1]=(Xr[0]-Xr[1])/2;
      Ri[1]=(Xi[0]-Xi[1])/2;
      Rr[0]=(Xr[0]+Xr[1])/2;
      Ri[0]=(Xi[0]+Xi[1])/2;
    }
  else
    {
      numt = pvm_spawn("fft",&fargv1[0],PvmTaskDefault,"",1,&lstid);
      if (numt != 1)
{
  printf("could not spawn left son\n");
```

```
  exit(1);
}
      numt = pvm_spawn("fft",&fargv2[0],PvmTaskDefault,"",1,&rstid);
      if (numt != 1)
{
  printf("could not spawn right son\n");
  exit(1);
}
      bufid = pvm_recv(lstid, RESULT_TAG);
      pvm_upkfloat(&T1[0],(argc-1)/2,1);
      bufid = pvm_recv(rstid, RESULT_TAG);
      pvm_upkfloat(&T2[0],(argc-1)/2,1);
      for (i=0;i<(argc-1)/4;i++)
{
  wr=cos(-4*PI*i/(argc-1));
  wi=sin(-4*PI*i/(argc-1));
  Rr[i]=(T1[i]+wr*T2[i]-wi*T2[i+(argc-1)/4])/2;
  Ri[i]=(T1[i+(argc-1)/4]+wr*T2[i+(argc-1)/4]+wi*T2[i])/2;
  Rr[i+(argc-1)/4]=(T1[i]-wr*T2[i]+wi*T2[i+(argc-1)/4])/2;
  Ri[i+(argc-1)/4]=(T1[i+(argc-1)/4]-wr*T2[i+(argc-1)/4]-wi*T2[i])/2;
}
    }
  pvm_initsend(PvmDataDefault);
  pvm_pkfloat(&Rr[0],(argc-1)/2,1);
  pvm_pkfloat(&Ri[0],(argc-1)/2,1);
  pvm_send(ptid,RESULT_TAG);
  pvm_exit();
  exit(0);
}
```

Chapitre 6

Compléments PVM

Dans ce chapitre, on présente quelques autres fonctionnalités avancées de PVM, en particulier l'organisation en "grappes" de processus, le contrôle dynamique de la machine virtuelle PVM, les signaux distribués et les mécanismes de tolérance aux pannes.

6.1 Les groupes de processus

Un groupe est un ensemble de processus,
- auquel on peut ajouter ou enlever dynamiquement de nouveaux membres (`pvm_joingroup`, `pvm_lvgroup`),
- dans lequel on peut gérer des processus individuels (`pvm_getinst`, `pvm_gettid`),
- dont on peut synchroniser tous les membres (`pvm_barrier`),
- auxquels on peut "broadcaster" un message (`pvm_bcast`) ou répartir un ensemble de données (`pvm_scatter`, etc.),
- qui peuvent exécuter une même opération sur leurs données (`pvm_reduce`).

Il y a un intérêt particulier dans le cas d'une utilisation de XPVM (interface graphique de PVM): XPVM est un processus PVM, donc on ne peut plus distinguer la racine des processus utilisateurs par `PvmNoParent`. Une solution est de mettre les processus utilisateurs dans un groupe "utilisateur".

6.1.1 pvm_joingroup

Cette instruction permet d'ajouter la tâche qui a appelé `pvm_joingroup` au groupe qui a pour nom la chaîne de caractères `group`:

```
int inum;
inum=pvm_joingroup(char *group);
```

Elle renvoie son numéro "d'instance" dans le groupe. C'est un numéro d'ordre entier commençant à 0 et s'incrémentant à chaque nouveau membre. Il y a un cas d'Erreur si `inum <0`.

6.1.2 pvm_lvgroup

Cette instruction retire le processus appelant du groupe `group`:

```
int info;
info=pvm_lvgroup(char *group);
```

Il y a un cas d'Erreur si `info <0`.

6.1.3 pvm_gsize

Cette instruction retourne le nombre de membres (à l'instant de l'exécution!) du groupe `group`:

```
int size;
size=pvm_gsize(char *group);
```

Il y a un cas d'<u>Erreur</u> si `info <0`.

6.1.4 pvm_gettid

Cette instruction renvoie l'identificateur de tâche du processus qui a pour numéro d'ordre `inum` dans le groupe `group`:

```
int tid;
tid=pvm_gettid(char *group,int inum);
```

Cela permet ainsi d'utiliser les primitives d'envoi et de réception de messages qui ont besoin de l'identificateur de tâche de l'expéditeur ou du destinataire. Il y a un cas d'<u>Erreur</u> si `tid <0`.

6.1.5 pvm_getinst

C'est le contraire de `pvm_gettid`: renvoie le numéro d'ordre dans le groupe `group` correspondant à l'identificateur de tâche `tid`:

```
int inum;
inum=pvm_getinst(char *group,int tid);
```

Il y a un cas d'<u>Erreur</u> si `inum <0`.

6.1.6 pvm_barrier (barrière de synchronisation)

Cette instruction bloque le processus exécutant `pvm_barrier` jusqu'à ce que `count` membres du groupe `group` se soient également bloqués:

```
int info;
info=pvm_barrier(char *group,int count);
```

Cela permet de synchroniser tout un groupe. `count` permet de gérer le fait que d'autres processus ont pu s'ajouter au groupe pendant l'attente. Il y a un cas d'<u>Erreur</u> si `info <0`.

6.1.7 pvm_bcast

Cette instruction "Broadcast" le tampon d'envoi précedemment rempli par des `pvm_pk...` vers tous les membres du groupe `group`, avec le tag `msgtag`:

```
int info;
info=pvm_bcast(char *group,int msgtag);
```

Il y a un cas d'<u>Erreur</u> si `info <0`.

6.1.8 Exemple

On désire calculer les sommes des 100 premiers entiers, de leurs carrés et de leurs cubes par décomposition fonctionnelle.

```
#include <stdio.h>
#include "pvm3.h"
#define NG      9    /* taille du groupe = NG+1 */
#define GROUP   "GR" /* nom du groupe */
#define ROOT    0    /* index du premier membre du groupe */
#define COUNT   10   /* Taille des sommes partielles */
#define N       100  /* (NG+1) x COUNT */
#define MSGTAG  1
#define ANYTASK -1

int main()
{
int gid, tids[NG], TG[N], TL[COUNT], i; long S[3];
/* Lancer la tache sous PVM */
if (pvm_mytid() < 0) { pvm_perror(); exit(-1); }
/* Rejoindre le groupe de nom GROUP  */
if ((gid = pvm_joingroup(GROUP)) < 0)
{
  pvm_exit(); exit(-1);
}
 if ( gid == ROOT ) {
   /* La premiere tache lance les autres membres... */
   if ( pvm_spawn("Sommes",NULL,PvmTaskDefault,NULL,NG,tids)<NG)
     {
       pvm_exit(); exit(-1);
     }
   /* ... et initialise les donnees a diffuser */
   for (i=0; i<N; i++) TG[i] = i;
 }
 /* Apres synchronisation, diffuser les donnees */
 pvm_barrier(GROUP, NG+1);
 if ( gid == ROOT )
   {
     pvm_scatter(&TL[0],&TG[0],COUNT,PVM_INT,MSGTAG,GROUP,ROOT);
   } else {
     pvm_scatter(&TL[0],NULL,COUNT,PVM_INT,MSGTAG,GROUP,ROOT);
   }
 /* Calculer les sommes partielles locales */
 S[0] = S[1] = S[2] = 0;
 for (i=0; i<COUNT; i++)
   {
     S[0] = S[0] + TL[i];
     S[1] = S[1] + TL[i]*TL[i];
     S[2] = S[2] + TL[i]*TL[i]*TL[i];
   }
 /* Sommer globalement les sommes partielles locales */
 pvm_reduce(PvmSum,&S[0],3,PVM_LONG,MSGTAG,GROUP,ROOT);
 /* Quitter le groupe */
 pvm_barrier(GROUP, NG+1);
 pvm_lvgroup(GROUP);
```

```
 /* Resultat */
 if ( gid == ROOT )
   {
     printf("1   + ... + %d   = %d\n", N, S[0]);
     printf("1^2 + ... + %d^2 = %d\n", N, S[1]);
     printf("1^3 + ... + %d^3 = %d\n", N, S[2]);
   }
 pvm_exit(); exit(0);
}
```

6.2 Contrôle de la machine PVM

6.2.1 pvm_addhosts

Cette instruction ajoute les `nhost` hôtes du tableau de chaînes `hosts` à la machine PVM:

```
int info;
info=pvm_addhosts(char **hosts,int nhost,int *infos);
```

Le tableau d'entiers `infos` contient l'état de l'ajout pour chaque hôte. Il y a un cas d'Erreur pour l'hôte numéro `i` si `infos[i] <0`. `info` retourne le nombre d'hôtes effectivement ajoutés.

6.2.2 pvm_delhosts

Cette instruction enlève les `nhost` hôtes du tableau de chaînes `hosts` à la machine PVM:

```
int info;
info=pvm_delhosts(char **hosts,int nhost,int *infos);
```

Le tableau d'entiers `infos` contient l'état de l'enlèvement pour chaque hôte. Il y a un cas d'Erreur pour l'hôte numéro `i` si `infos[i] <0`. `info` retourne le nombre d'hôtes effectivement enlevés.

6.3 Signaux

On peut gérer des interruptions avec la librairie PVM, étendant en cela la gestion des signaux C standards à un ensemble de processus.

6.3.1 Rappel: les signaux en C

L'instruction `void signal(int, void (*)(int))` permet de définir ce que le programme doit faire en cas d'exception. Le premier argument est le type d'interruption, le deuxième est la fonction à appeler quand l'interruption arrive ("handler"). Les types d'interruptions sont définis dans `sys/signal.h`:

```
#define SIGHUP  1 /* hangup */
#define SIGINT  2 /* interrupt (rubout) */
#define SIGQUIT 3 /* quit (ASCII FS) */
#define SIGILL  4 /* illegal instruction (not reset when caught) */
#define SIGTRAP 5 /* trace trap (not reset when caught) */
#define SIGIOT  6 /* IOT instruction */
#define SIGABRT 6 /* used by abort, replace SIGIOT in the future */
```

```
#define SIGEMT  7 /* EMT instruction */
#define SIGFPE  8 /* floating point exception */
#define SIGKILL 9 /* kill (cannot be caught or ignored) */
#define SIGBUS  10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS  12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD  18 /* child status change */
#define SIGCHLD 18 /* child status change alias POSIX */
#define SIGPWR  19 /* power-fail restart */
#define SIGWINCH 20 /* window size change */
#define SIGURG  21 /* urgent socket condition */
#define SIGPOLL 22 /* pollable event occurred */
#define SIGIO   SIGPOLL /* socket I/O possible (SIGPOLL alias) */
#define SIGSTOP 23 /* stop (cannot be caught or ignored) */
#define SIGTSTP 24 /* user stop requested from tty */
#define SIGCONT 25 /* stopped process has been continued */
#define SIGTTIN 26 /* background tty read attempted */
#define SIGTTOU 27 /* background tty write attempted */
#define SIGVTALRM 28 /* virtual timer expired */
#define SIGPROF 29 /* profiling timer expired */
#define SIGXCPU 30 /* exceeded cpu limit */
#define SIGXFSZ 31 /* exceeded file size limit */
#define SIGWAITING 32 /* process's lwps are blocked */
#define SIGLWP  33 /* special signal used by thread library */
...
```

Par exemple:

```
/* sig1clavier - P. Cousot            */
/* intercepter les signaux du clavier */
/* arreter par ^\                     */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int i = 0;
/* traitement des signaux */
void on_SIG( int code ) {
  fprintf(stdout, "\nSignal (code = %d,i = %d)\n",code,i);
  fflush(stdout);
  if (code == SIGQUIT) exit(1);
}

int main( void ) {
  /* interception des signaux */
  signal(SIGINT, on_SIG);  /* ^C                        */
  signal(SIGTERM, on_SIG); /* kill PID si arriere plan */
  signal(SIGQUIT, on_SIG); /* ^\                        */
```

```
  /* boucle infinie */
  for(;i<100000000;i++ ){
    if ((i % 100000) == 0)
      {
        putchar('.');
        fflush(stdout);};
      }
  fprintf(stdout, "Fin\n");
}
```

Qui s'exécute comme suit:

```
eider:/home/eider/goubault 62 % a.out
.................^C
Signal (code = 2, i = 1674042)
......................^\
Signal (code = 3, i = 3878990)
eider:/home/eider/goubault 63 %
```

6.3.2 Signaux sous PVM

pvm_sendsig

Cette instruction envoie le signal d'interruption qui a pour numéro `signum` au processus `tid`:

```
int info;
info=pvm_sendsig(int tid,int signum);
```

Il y a un cas d'Erreur si `info <0`.

Exemple PVM

On considère le programme maître suivant, qui lance un esclave `sig2` puis lui envoie `SIGTERM` et `SIGQUIT`:

```
#include <stdio.h>
#include <pvm3.h>
#include <signal.h>

main()
{
  int mtid,ctid;
  if ((mtid=pvm_mytid()) < 0)
    {
      pvm_perror("pvmd?");
      pvm_exit();
    }
  fprintf(stdout,"I am spawning a child\n");
  pvm_spawn("sig2",(char **) 0,PvmTaskDefault,(char *) 0,1,&ctid);
  fflush(stdout);
  sleep(1);
  fprintf(stdout,"I am sending SIGTERM\n");
  pvm_sendsig(ctid,SIGTERM);
  fflush(stdout);
  sleep(2);
```

```
  fprintf(stdout,"I am sending SIGQUIT\n");
  pvm_sendsig(ctid,SIGQUIT);
  fflush(stdout);
  sleep(1);
  pvm_exit();
  exit(1);
}
```

L'esclave "sig2" essaie d'intercepter les signaux envoyés par le maître:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <pvm3.h>

int i = 0;
/* traitement des signaux */
void on_SIG( int code ) {
  fprintf(stdout, "\nSignal (code=%d,i=%d)\n",code,i);
  fflush(stdout);
  if (code == SIGQUIT) { pvm_exit(); exit(1);}
}

int main( void ) {
  /* interception des signaux */
  signal(SIGINT, on_SIG);  /* ^C                       */
  signal(SIGTERM, on_SIG); /* kill PID si arriere plan */
  signal(SIGQUIT, on_SIG); /* ^\                       */
  /* boucle infinie */
  for(;i<100000000;i++ ){
    if ((i % 100000) == 0)
      {
putchar('.');
fflush(stdout);};
  }
  fprintf(stdout, "Fin\n");
  fflush(stdout);
  pvm_exit();
  exit(1);
}
```

L'exécution du programme est alors:

```
pvm> spawn -1 -> sig1
[4]
1 successful
t40013
pvm> [4:t40013] I am spawning a child
[4:t40013] I am sending SIGTERM
[4:t40014] .................
[4:t40014] Signal (code = 15, i = 1795243)
[4:t40013] I am sending SIGQUIT
[4:t40014] ..............................
[4:t40014] Signal (code = 3, i = 4974635)
[4:t40014] EOF
```

```
[4:t40013] EOF
[4] finished

pvm> jobs
pvm>
```

6.4 Gestion des erreurs

On peut connaître le type d'erreur PVM d'une commande. On peut également être averti de la mort d'une machine composant la machine PVM ou d'un processus.

6.4.1 pvm_perror

Cette instruction affiche un message d'erreur:

```
int info;
info=pvm_perror(char *msg);
```

6.4.2 pvm_notify

Cette instruction demande à PVM de notifier l'appelant de l'apparition de certains événements: `what` peut être `PvmTaskExist` (une tâche quitte PVM), `PvmHostDelete` (une des machines de la machine PVM n'est plus là), `PvmHostAdd` (une nouvelle machine est ajoutée à la machine PVM). A part le dernier événement, les processus auxquels on s'intéresse sont dans le tableau `tids`. `cnt` ne sert que dans le cas `PvmHostAdd`. Quand un de ces événements apparaît, un certain nombre de messages avec le tag `msgtag` sont envoyés à l'appelant:

```
int info;
info=pvm_notify(int what,int msgtag,int cnt,int *tids);
```

Il y a un cas d'Erreur si `info <0`.

Par exemple (tiré de la documentation PVM), le maître:

```
/* Failure notification example
   Demonstrates how to tell when a task exits */
#include <stdio.h>
#include <pvm3.h>
/* Maximum number of children this program will spawn */
#define MAXNCHILD   20
/* Tag to use for the task done message */
#define TASKDIED        11

int main(int argc, char* argv[])
{
  int ntask,info,mytid,child[MAXNCHILD],i,deadtid,tid;
  ntask = 3;
  mytid = pvm_mytid();
  if (mytid < 0)
    {
      pvm_perror(argv[0]);
      return -1;
    }
  /* find out how many tasks to spawn */
  if (argc == 2) ntask = atoi(argv[1]);
```

```
  /* make sure ntask is legal */
  if ((ntask < 1) || (ntask > MAXNCHILD))
    {
      pvm_perror("wrong number of tasks");
      pvm_exit(); return 0;
    }
  /* spawn the child tasks */
  info = pvm_spawn("notify_child",(char**)0,PvmTaskDefault,(char*)0,
                                                   ntask,child);
  printf("I have spawned %d tasks\n",info);
  fflush(stdout);
  if (info != ntask)
    {
      pvm_perror("Pb in spawn");
      pvm_exit(); return -1;
    }
  for (i = 0; i < ntask; i++) printf("t%x\t",child[i]);
  putchar('\n');
  fflush(stdout);
  /* ask for notification when child exits */
  info = pvm_notify(PvmTaskExit, TASKDIED, ntask, child);
  if (info < 0)
    {
      pvm_perror("notify"); pvm_exit();
      return -1;
    }
  /* reap the middle child */
  info = pvm_kill(child[ntask/2]);
  if (info < 0)
    {
      pvm_perror("kill"); pvm_exit();
      return -1;
    }
  /* wait for the notification */
  info = pvm_recv(-1, TASKDIED);
  if (info < 0)
    {
      pvm_perror("recv"); pvm_exit();
      return -1;
    }
  info = pvm_upkint(&deadtid, 1, 1);
  if (info < 0) pvm_perror("calling pvm_upkint");
  /* should be the middle child */
  printf("Task t%x has exited.\n", deadtid);
  printf("Task t%x is middle child.\n", child[ntask/2]);
  fflush(stdout);
  pvm_exit();
  return 0;
}
```

Les esclaves “notify_child”:

```
#include <stdio.h>
#include <pvm3.h>
```

```
main()
{   /* i'm a child */
  printf("I am not root and my parent is t%x\n",pvm_parent());
  sleep(10);
  pvm_exit();
  return 0; }
```

L'exécution du programme est la suivante:

```
pvm> spawn -1 -> notify 10
[2]
1 successful
t40006
pvm> [2:t40006] I have spawned 10 tasks
[2:t40006] t40007       t40008  t40009  t4000a  t4000b
           t4000c  t4000d  t4000e  t4000f  t40010
[2:t4000c] EOF
[2:t40006] Task t4000c has exited.
[2:t40006] Task t4000c is middle child.
[2:t40006] EOF
[2:t40007] I am not root and my parent is t40006
[2:t40007] EOF
[2:t40008] I am not root and my parent is t40006
[2:t40008] EOF
[2:t4000a] I am not root and my parent is t40006
[2:t4000a] EOF
[2:t40009] I am not root and my parent is t40006
[2:t40009] EOF
[2:t4000e] I am not root and my parent is t40006
[2:t4000e] EOF
[2:t4000b] I am not root and my parent is t40006
[2:t4000b] EOF
[2:t4000d] I am not root and my parent is t40006
[2:t4000d] EOF
[2:t40010] I am not root and my parent is t40006
[2:t40010] EOF
[2:t4000f] I am not root and my parent is t40006
[2:t4000f] EOF
[2] finished
```

Chapitre 7

Ecriture de programmes parallèles

Pour l'écriture de code parallèle on peut soit paralléliser un code séquentiel (automatiquement), soit ordonnancer des tâches parallèles à la main.

La difficulté est que l'efficacité dépend de l'architecture de la machine cible et de l'algorithmique du problème.

La morale est qu'il faut bien réfléchir à l'algorithmique sous-jacente avant de se lancer dans l'ordonnancement de tâches, l'optimisation du code etc. On pourra se reporter à [LER92] pour plus de détails.

7.1 Méthodes classiques

Dans le génie logiciel parallèle traditionnel, les notions importantes sont celles de *tâche*, *processus* et *processeur*. Une tâche est un ensemble d'instructions séquentielles qui représentent de façon logique une petite étape de calcul, insécable. Un processus est aussi une abstraction logique: c'est un ensemble de tâches organisées de façon séquentielle.

On obtient un code parallèle à partir d'un algorithme, en procédant par un certain nombre d'étapes, que l'on nomme souvent partitionnement, coordination et enfin affectation. Tout ceci va être essentiellement décrit à travers un certain nombre d'exemples.

7.1.1 Partitionnement

Le partitionnement consiste à essayer de trouver des candidats à la parallélisation. On commence par essayer de décomposer le problème en un certain nombre de tâches. Puis on affecte les tâches à un ensemble de processus. En général, une tâche effectue une communication ou une synchronisation. Quand la tâche est petite (en temps d'exécution), on parle de parallélisme à *grain fin*. Quand la tâche augmente de taille, on parle de *grain moyen* puis de *gros grain*: c'est le cas quand un processus contient une seule tâche et une seule synchronisation avec les autres processus.

Cela peut se faire par *décomposition de domaines*: diviser les données, puis les calculs correspondants. Cela peut aussi se faire par *décomposition fonctionnelle*: diviser les calculs, puis les données correspondantes.

Exemple 1 Considérons par exemple un schéma explicite d'ordre 1 pour résoudre une forme de l'équation des ondes bidimensionnelle. Supposons que l'on dispose de p processeurs, et que l'on ait discrétisé le problème par une grille de $n \times n$ points

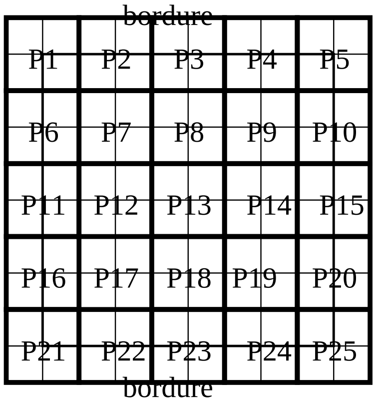


FIG. 7.1 – *Partionnement d'un schéma explicite 2D*

comme indiqué sur la figure 7.1 [1]. Le calcul de la valeur du champ de pression u au point de coordonnées (i, j) $(1 \leq i, j \leq n-1)$, à l'itération $t+1$, $u^{t+1}(i, j)$ dépend uniquement des valeurs de ses quatre voisins $u^t(i-1, j)$, $u^t(i, j-1)$, $u^t(i+1, j)$ et $u^t(i, j+1)$ ainsi que de $u^t(i, j)$.

Alors, chaque processeur P_i $(1 \leq i \leq p)$ aura une portion, comme indiqué sur la figure 7.1, de la grille, et n'aura à se synchroniser, et à échanger de l'information que sur le périmètre de chaque domaine.

Un point important à toujours garder en tête quand il s'agit de performances, est la *loi d'Amdahl.*

Exemple 1 suite Considérons par exemple un calcul simple sur la grille $n \times n$ avec p processeurs, évoqués précédemment. Supposons que ce calcul soit composé de deux sous-calculs; un premier qui agit sur chaque point de la grille, et ce de façon complètement indépendante de ses voisins, et un deuxième, qui additionne toutes les valeurs à chaque point de la grille. Par exemple, la première partie est un calcul de carré, la deuxième une somme, le tout faisant un calcul d'énergie au carré (qui peut être utile pour la validation numérique du programme, ou du schéma). La première partie prendra donc un temps de $\frac{n^2}{p}$. Une façon de programmer la deuxième partie est de séquentialiser entièrement le calcul de la somme globale, qui va alors prendre un temps de n^2 (indépendant de p!). Le temps de l'algorithme séquentiel aurait été de $2n^2$ alors que le temps mis par cet algorithme est de $\frac{n^2}{p} + 2n^2$, donc l'accélération obtenue est d'au plus $\frac{2p}{p+1}$, c.a.d. au plus de 2 quel que soit le nombre de processeurs que l'on ait à disposition.

C'est une conséquence simple de la loi d'Amdahl qui s'énonce comme suit. Si on a un calcul dont s pourcent est séquentiel, et qui s'exécute sur p processeurs, alors l'accélération du calcul complet par rapport à un calcul séquentiel sera au maximum de,

$$\frac{1}{s + \frac{1-s}{p}}$$

c'est-à-dire que même dans le cas où on dispose d'une infinité de processeurs, l'accélération maximale sera de $\frac{1}{s}$. Conséquence: même si on parallélise 80 pourcent

1. On remarquera à ce propos que l'on a une bordure de points que l'on ne calcule pas, mais qui est présente pour forcer les conditions aux limites de façon simple.

d'un code (le reste étant séquentiel), on ne pourra jamais dépasser, quel que soit la machine cible, une accélération d'un facteur 5!

Exemple 1 suite En fait, on pouvait faire bien mieux dans notre exemple. Tous les processeurs peuvent, en même temps, calculer une somme partielle locale. Puis une somme séquentielle de p sommes partielles, prenant un temps[2] de p. L'accélération est maintenant de $\frac{2pn^2}{2n^2+p}$ qui est presque linéaire en le nombre de processeurs (quand n est grand par rapport à p).

Décomposition de données - Exemple 2

On a vu la transformée de Fourier rapide à la section 5.2.2. Ici on va examiner des parallélisations possibles, dans le cas du calcul bidimensionnel.

Le réseau que nous considérerons dans toute la suite est composé de N ($N \geq 2$) stations nommées $P_1, P_2, \cdots, P_N$. La donnée de départ est une matrice complexe carrée $M = (M_{i,j})_{1 \leq i,j \leq K}$ de taille $K \times K$ (N divise K), entièrement stockée dans la mémoire locale de P_1. On veut construire des algorithmes qui répartissent au mieux le travail sur les $P_1, P_2, \cdots, P_N$ de telle façon qu'à la fin de leur exécution, la transformée de Fourier $\hat{M}$ se trouve stockée dans la mémoire locale de P_1.

On considérera que les performances de nos algorithmes ne dépendront que des données suivantes :

- le temps d'un envoi d'un message, sans compter l'attente éventuelle d'un partenaire pour la communication, est le débit γ multiplié par la taille en nombre de complexes du tableau envoyé plus le temps de latence δ.
- le temps d'un cycle sur P_i ($1 \leq i \leq N$) est de α_i (une constante).

On examine d'abord l'algorithme A suivant, inspiré de la solution naturelle en mode SIMD.

P_1 (le maître) crée des processus "*esclave*" sur chacune des autres stations (P_i, $2 \leq i \leq N$) puis envoie à chacune d'entre elles le $i$ème paquet de $\frac{K}{N}$ lignes de la matrice M. P_1 exécute ensuite fft sur les $\frac{K}{N}$ premières lignes de M puis se met en attente des transformées de Fourier exécutées sur chaque autre paquet de $\frac{K}{N}$ lignes de M par les esclaves. Une fois reçues et stockées dans la matrice N, P_1 envoie à chaque esclave P_j le $j$ème paquet de $\frac{K}{N}$ colonnes sur lesquelles P_j effectue encore une fois fft. Enfin P_1, qui a calculé pendant ce temps la transformée du premier paquet de colonnes, réceptionne les transformées effectuées par les autres stations et les stocke dans $\hat{M}$.

Le pseudo-code pour le maître est,

```
struct complex M[K][K];
int tids[N-1];

fft_une_dimension()
{ int i;
  for(i=1; i<=N-1; i++)
   {
    pkcmplx(&M[i*K/N][0],K*K/N,1);
    pvm_send(tids[i-1],0);
   }
  for(i=0; i<K/N; i++) fft(M[i],K);
  for(i=1; i<=N-1; i++)
   {
```

2. En fait, il s'agit d'une opération de *réduction*, qui existe sous PVM par exemple, et qui est implémentée par un algorithme diviser-pour-régner, prenant un temps de $\log_2 p$.

```
    pvm_recv(tids[i-1],0);
    upkcmplx(&M[i*K/N][0],K*K/N,1);
   }
}

transpose()
{ int i,j;
  for(i=0; i<K; i++)
   for(j=0; j<i; j++)
    M[j][i]=M[i][j];
}

maitre()
{ pvm_spawn("esclave",NULL,0,0,N-1,tids);
 fft_une_dimension();
 transpose();
 fft_une_dimension();
 transpose();
}
```

Le pseudo-code pour les esclaves est,

```
fft_une_passe()
{ int i,tid=pvm_parent();
  complex T[K/N][K];
  pvm_recv(tid,0);
  upkcmplx(T,K*K/N,1);
  for(i=0; i<K/N; i++) fft(T[i],K);
  pkcmplx(T,K*K/N,1);
  pvm_send(tid,0);
}

esclave()
{
  fft_une_passe();
  fft_une_passe();
}
```

Décomposition fonctionnelle - Exemple 3

On peut remarquer que, en supposant $i, j \leq \frac{K}{2}$, on a:

$$
\begin{aligned}
\hat{M}_{ij} &= \frac{1}{K^2}\sum_{k=0}^{K}\sum_{l=0}^{K} M_{kl}e^{-\frac{2i\pi}{K}ik}e^{-\frac{2i\pi}{K}jl} \\
&= \frac{1}{4}\left(\frac{1}{K/4}\sum_{k \text{ pair}, l \text{ pair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl} + \frac{1}{K/4}\sum_{k \text{ impair}, l \text{ pair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl} + \right. \\
&\quad \left. \frac{1}{K/4}\sum_{k \text{ pair}, l \text{ impair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl} + \frac{1}{K/4}\sum_{k \text{ impair}, l \text{ impair}} M_{kl}e^{-\frac{2i\pi}{K}ikjl}\right) \\
&= \frac{1}{4}\left(\frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k,2l}e^{-\frac{2i\pi}{K}i2kj2l} + \frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k+1,l}e^{-\frac{2i\pi}{K}i(2k+1)j2l}\right.
\end{aligned}
$$

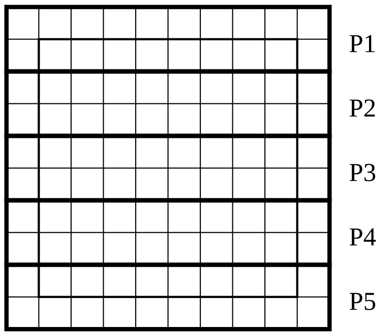


FIG. 7.2 – *Une autre organisation des domaines pour un schéma explicite simple*

$$
\begin{aligned}
& +\frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k,2l+1}e^{-\frac{2i\pi}{K}i2kj(2l+1)} \\
& \left. +\frac{1}{K/4}\sum_{k=0}^{\frac{K}{2}}\sum_{l=0}^{\frac{K}{2}} M_{2k+1,2l+1}e^{-\frac{2i\pi}{K}i(2k+1)j(2l+1)}\right) \\
= \; & \frac{1}{4}\left(\hat{M}_{ij}^{pp} + e^{-\frac{2i\pi}{K}i}\hat{M}_{ij}^{ip} + e^{-\frac{2i\pi}{K}j}\hat{M}_{ij}^{pi} + e^{-\frac{2i\pi}{K}(i+j)}\hat{M}_{ij}^{ii}\right)
\end{aligned}
$$

Le principe de la décomposition fonctionnelle est le même que celui dans le cas de la FFT unidimensionnelle (section 5.2.2): les esclaves sont créés en parallèle jusqu'à l'étape h où les feuilles calculent la FFT séquentiellement. Dans une deuxième phase les esclaves remontent la FFT grâce à la propriété que l'on vient de prouver. On peut cependant minimiser le temps inactif de chaque station en ne "spawnant" jamais la quatrième branche mais en conservant cette exécution locale.

7.1.2 Coordination

Elle peut s'organiser de façons très différentes selon les besoins: locale où globale (avec un centralisateur ou pas), décidée de façon statique ou dynamique, et enfin synchrone ou asynchrone (communications bloquantes ou pas).

Deux principes essentiels sont d'assurer un maximum de *recouvrement calcul et communication* dans le cas où les communications sont asynchrones (ou non-bloquantes) et d'assurer une certaine localité des données afin de limiter le nombre et la taille des communications.

Exemple 2 suite Par exemple, le schéma explicite de la section 7.1.1 a un temps de communication par processeur de l'ordre de $\frac{4n}{\sqrt{p}}$ (le périmètre de chaque domaine) pour un temps de calcul par processeur de l'ordre de $\frac{n^2}{p}$. Par contre, ayant quatre voisins en général avec qui communiquer, la coordination globale à tendance à beaucoup attendre les retardataires voisins. On pourrait en fait faire ici des communications asynchrones (dont les réceptions seraient affectées à une bordure de chaque domaine, sans synchronisation intempestive).

Une autre solution est de changer la géométrie des domaines. Par exemple, on pourrait subdiviser la grille 2D en bandes horizontales comme indiqué sur la figure 7.2 La taille des communications par processeurs est alors plus grande en général et est indépendante de p (de l'ordre de $2n$). Par contre, n'ayant qu'au maximum deux voisins, il est possible de moins synchroniser certains calculs.

Les temps de communication pour la décomposition de domaine de la figure 7.1 peuvent être décomptés comme suit. Le temps de communication du maître est $2(N-1)(\frac{K^2}{N}\gamma+\delta)$ pour les envois et $2(N-1)\frac{K^2}{N}\alpha_1$ pour les réceptions. Le temps de communication d'un esclave est de même $2(\frac{K^2}{N}\gamma+\delta)$ et $2\frac{K^2}{N}\alpha_i$ respectivement.

L'esclave numéro P_i attend $i-1$ compactages et envois pour la réception des données lors de sa première passe. Il attend en outre la transposition et $N-i+1$ décompactages et réceptions lors de sa deuxième passe.

Le maître n'attend pas pour la réception des données des esclaves dès lors que les esclaves sont aussi rapides que le maître et que le nombre d'esclaves est supérieur ou égal à deux. Par contre si les α_i étaient différents, le maître devrait attendre les esclaves les plus lents. Donc, si on suppose que les esclaves sont aussi rapides que le maître, le temps de calcul pour cet algorithme est le temps de calcul du maître plus son temps de communication.

- `spawn` : τ
- `transpose` : $(K^2-2K)\alpha_1$
- `fft` : $2\frac{K}{N}\theta(KlogK)\alpha_1$
- envois : $2(N-1)(\frac{K^2}{N}\gamma+\delta)$
- réceptions : $2(N-1)\frac{K^2}{N}\alpha_1$
- compactage et décompactage : $4(N-1)\frac{K^2}{N}\alpha_1$

Si les α_i étaient différents, ce serait la plus lente des stations qui imposerait la cadence. On pourrait modifier l'algorithme en connaissant les coefficients, de telle façon à répartir plus de lignes (respectivement colonnes) sur les stations les plus rapides. Plus précisément P_i devrait traiter $\frac{1/\alpha_i}{\sum 1/\alpha_i}K$ lignes.

Exemple 2 amélioré La communication recouvre donc bien les calculs dans l'exemple précédent. La coordination était globale. On peut "localiser" un peu celle-ci, afin que chaque esclave gère lui-même l'envoi de ses parties de colonnes directement (sans passer par P_1) aux stations correspondantes.

Une fois la transformée en ligne calculée sur une station P_i (paquet de ligne p_i), on peut exécuter les *pvm_send* des paquets de colonnes c_j vers les $P_1, \cdots, P_{i-1}$, puis dans un ordre quelconque les *pvm_recv* des paquets de colonnes c_i appartenant aux stations $P_1, P_2, \cdots, P_{i-1}, P_{i+1}, \cdots P_N$ et les *pvm_send* restants (colonnes $c_{i+1}, \cdots, c_N$). Ceci assure qu'il n'y aura pas d'interblocage.

On y gagne en temps de communication, $(N-1)(\frac{K^2}{N}\gamma+\delta)$ pour les envois et $(N-1)\frac{K^2}{N}\alpha$ pour les réceptions, plus en temps de compactage/décompactage : $2(N-1)\frac{K^2}{N}\alpha$.

Exemple 3 suite Dans l'exemple 3 de la section 7.1.1, le temps de communication de la décomposition fonctionnelle (en "spawnant" quatre fils à chaque fois) est de $2(h-1)[K^2(\gamma+\alpha)+\delta]$. Celui où on ne "spawne" que trois fils est de $2h[3\frac{K^2}{4}(\gamma+\alpha)+\delta]$. Il est supérieur à celui de l'algorithme par décomposition de données dès que $h \geq 3$.

Le temps de calcul de la procédure séquentielle `fft` pour P_i aux feuilles (dont la taille est $\frac{K}{2^{h-1}}$) est

$$2\frac{K}{2^{h-1}}\theta(\frac{K}{2^{(h-1)}}log(\frac{K}{2^{(h-1)}}))\alpha_i$$

soit un ratio favorable de $\frac{1}{2}logN$ pour la décomposition fonctionnelle vis-à-vis de l'algorithme par décomposition de données (de $\frac{3}{8}logN$ si on avait "spawné" trois fils seulement). Il faut cependant encore ajouter la remontée du calcul de la FFT, $\theta(\frac{K^2}{4^n})\alpha_i$ pour un esclave P_i à la profondeur $0 \leq n < h$.

7.1.3 Equilibrage de charges et tolérance aux pannes

Supposons ici que les α_i (pour l'algorithme de FFT bidimensionnel) puissent être très différents les uns des autres et ne soient connus par aucun des P_i. Autorisons nous également pour un nombre de stations $t < N-1$ parmi les P_2 à P_N un α_i égal à plus l'infini, modélisant ainsi la panne juste après la connection à *pvmd* (panne immédiate). Dans ce cas, on supposera qu'il n'y a aucun moyen pour aucune des stations de détecter la panne d'une autre des stations.

Quand on ne connaît pas, et quand on ne peut pas déterminer les processeurs morts, ou leurs vitesses relatives, l'allocation statique des tâches que l'on a réalisé aussi bien pour l'exemple 2 que pour l'exemple 3 de 7.1.1 n'est pas bonne. Si un des α_i est égale à $+\infty$ alors le maître se bloque, et si les α_i sont très différents cela occasionne beaucoup d'attente.

On donne ici deux méthodes simples pour résoudre le problème. On conseille, pour plus de détails, la lecture de [Lyn96].

Une solution par "timeout" Le principe est de décider qu'après le temps t les stations qui n'ont pas répondu sont mortes et alors de les ignorer. Le problème est de choisir t. Si t est tel que l'on en déduise que toutes sont mortes, on double t et on recommence, ainsi de suite. Comme on a supposé que toutes les stations ne pouvaient pas mourir, il existe une borne au-dessus de laquelle l'une d'entre elles répondra.

Une solution par une architecture "client/serveur" L'idée est de simuler une mémoire partagée (contenant essentiellement M) par un "serveur" (P_1). Les accès par les autres stations ("clients") se font à travers des communications asynchrones : des messages avec le TAG "write" correspondent à une demande faite par un client d'écrire son paquet (de lignes ou de colonnes) dans M, stockée localement dans la mémoire de P_1. Des messages avec le TAG "read" correspondent à une demande de lecture d'un paquet de lignes faite par un client au serveur P_1. On peut implémenter ce client/serveur en PVM par signaux ou par "active polling" [3].

La modification de l'algorithme par décomposition de données (section 7.1.1) est alors comme suit :

- En plus de M, P_1 gère une collection de drapeaux (variables booléennes) : un drapeau $d(l)$ (respectivement $d(c)$) est associé à chaque ligne l (respectivement à chaque colonne c) de M. Ces drapeaux sont utilisés pour savoir (et faire savoir) si une ligne (respectivement une colonne) est utilisée par un client. Les drapeaux sont initialement à faux.
- Chaque esclave demande la ligne 1, puis la ligne 2 etc. Le premier qui l'obtient fait son calcul de fft puis demande à réécrire dans M la ligne correspondante. Quand tous les drapeaux des lignes sont à vrai, les esclaves font la compétition pour calculer les transformées en colonne cette fois-ci.

Supposons d'abord que chacun des clients demande d'abord au serveur la valeur du drapeau de la ligne 1, de la ligne 2, etc. jusqu'à en trouver un à la valeur faux. A ce moment là, le client demande la lecture de la ligne correspondante (ce qui met automatiquement le drapeau correspondant à vrai). Le problème est d'avoir dissocié le test du drapeau, de sa modification (problème d'*atomicité*, et donc de *cohérence* de l'état du système): si on a un processus qui se glisse entre le test et la demande de lecture, pour faire lui-même le test, on peut avoir deux processeurs exécutant le même calcul.

3. Ou encore en utilisant les "*handler*" de PVM3.4, voir chapitre B, permettant d'exécuter une fonction (définie par le programmeur) au moment de la réception d'un message avec le bon TAG.

Supposons maintenant que les clients demandent au serveur la première ligne (ou colonne) dont le drapeau est à faux. Le serveur étant purement séquentiel, quelque soit la rapidité des différents clients, il répond aux requêtes dans un certain ordre (propriété de *séquentialisation* assurant la cohérence). Le serveur ne peut plus donner la même ligne (ou colonne) à deux clients. Les calculs sont donc répartis sans redondance.

Cette méthode résoud le problème des pannes immédiates, mais pas d'une panne qui pourrait se produire à n'importe quel moment du calcul. Un client qui tombe en panne après avoir lu une ligne (ou colonne) aura réservé le drapeau correspondant et finalement va bloquer le calcul global. Par contre en cas de pannes immédiates le calcul ira jusqu'à son terme si au moins un client reste actif, car les clients en panne immédiate ne peuvent pas lire de ligne (ou colonne).

La compétition entre les processeurs avantage en moyenne les plus rapides : cette méthode sera plus efficace quand les α_i sont différents, que celles utilisant un partitionnement statique (qui n'équilibre pas bien les charges).

7.1.4 Contention et Bande Passante

Dans le schéma explicite, exemple 1 de la section 7.1.1, chaque processeur doit envoyer un volume de données égal à son périmètre à chacun de ses 2, 3 ou 4 voisins (de l'ordre de $\frac{4n}{\sqrt{p}}$). Il est possible (en général quand on passe aux problèmes tridimensionnels), que la bande passante du réseau, ou du bus d'interconnexion soit dépassé par le volume de données ayant à transiter, entraînant ainsi une séquentialisation et un délai. Il peut s'avérer utile à ce moment là de changer la géométrie des domaines, ou de segmenter les envois de messages. La solution où les domaines sont des bandes horizontales est moins bonne de ce point de vue là car le périmètre, donc le volume de communication est de l'ordre de $2n$ (indépendant de p).

7.1.5 Agglomération

Le principe est d'essayer d'augmenter la granularité des calculs faits en parallèle, de répliquer les calculs ou les données (quand la communication des données ou des résultats est trop chère), et en somme d'éviter au maximum les communications. Par exemple, dans certains problèmes et sur certaines architectures, il peut être trop couteux d'echanger certaines données. On copie alors toutes les données (au démarrage) sur chaque station du réseau. Le problème est alors que l'on ne peut plus gérer des gros problèmes qui eux ont besoin de plus de mémoire qu'un ordinateur séquentiel peut adresser.

7.1.6 Affectation

C'est la phase où on essaie de déterminer comment sont réparties les processus sur les processeurs. Cela peut se faire de différentes manières, selon que l'on connaisse ou pas la topologie de communication (et les coûts associés). Le problème est un cas particulier du problème d'ordonnancement de tâches parallèles pour lequel on donne quelques pointeurs dans la section suivante.

Exemple 4 Prenons ici l'exemple classique de la multiplication de matrices.

En mémoire partagée le code suivant effectue une simple multiplication de matrices par blocs colonnes:

```
for (i=1;i<=n;i++)
 for (j=1;j<=n;j++)
  for (k=1;k<=n;k++)
```

```
  C_{i,j}=C_{i,j}+A_{i,k}B_{k,j};
```

On peut aussi organiser la localité des données par blocs rectangulaires. Le code devient alors:

```
for (j=1;j<=N;j++)
 for (k=1;k<=n;k++)
  C^{(j)}=C^{(j)}+A(1:n,k)B(k,1:n/N);
```

Les blocs seront alors répartis par le compilateur en général (ou avec l'aide de directives de compilation données par l'utilisateur) comme pour l'exemple 1.

Sur une machine distribuée, on peut écrire des algorithmes dépendant plus finement de la topologie de communication. Supposons ici une topologie de communication particulière: les processeurs sont reliés (de façon logique tout au moins) comme s'ils formaient une grille $N \times N$ de processeurs en forme de tore. L'algorithme de Cannon qui suit permet d'effectuer au mieux le calcul de la multiplication de deux matrices.

```
forall (i=0;i<=N-1;i++)
 left_shift_row A(i,*) by i;
forall (j=0;j<=N-1;j++)
 up_shift_column A(*,j) by j;
forall (k=1;k<=N;k++)
 forall (i=0;i<=N-1;i++)
  forall (j=0;j<=N-1;j++)
   C_{i,j}=C_{i,j}+A_{i,j}B_{i,j};
   left_shift_row A by 1;
   up_shift_column B by 1;
```

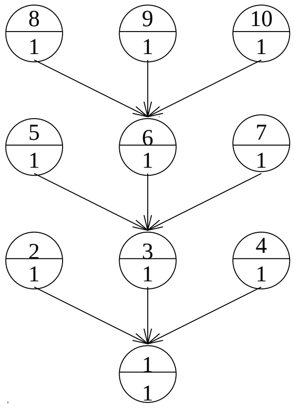
Considérons uniquement des matrices $n \times n$ sur un tore $N \times N$ de processeurs avec $n = kN$, k pair. Alors on peut voir que le nombre de messages envoyés en parallèle est $4N - 2$, le nombre de coefficients envoyés est $(\frac{n}{N})^2.(4N - 2)$ et le nombre d'opérations flottantes en parallèle est $\frac{2n^3}{N^2}$.

7.2 Ordonnancement de tâches parallèles

Il y a plusieurs options. On peut le faire au niveau local c'est à dire au niveau du microprocesseur (vectorialisation etc.). On peut aussi le faire de façon plus globale et essayer de répartir les tâches sur les processeurs. L'ordonnancement statique est une répartition avant exécution, alors que l'ordonnancement dynamique se fait pendant l'exécution.

Il y a plusieurs cadres dans lesquels on peut se poser le problème. Il faut savoir si on est intéressé par un environnement mono-application ou pas: il peut y avoir dans le cas contraire des problèmes de modélisation de l'attente de ressources partagées. Il faut également savoir si l'on veut pouvoir tenir compte d'éventuelles prémption ou pas: dans le premier cas on s'autorise à interrompre l'exécution d'une tâche et à la redémarrer à n'importe quel moment, ce qui complique singulièrement la tâche.

On se place pour la suite de cette section en mono-application, non-préemptif, statique, et global. Le modèle simplifié que nous allons utiliser est le suivant. On part d'un graphe pour la machine cible: $P = \{P_1, \cdots, P_m\}$ sont les processeurs, $(P_{i,j})$ est la matrice de connection entre les processeurs, (S_i) sont les vitesses de chacun des processeurs. On a également un graphe des tâches $(T, <, (D_{i,j}), (A_i), (W_j))$ avec $T = \{T_1, \cdots, T_n\}$ l'ensemble des tâches, $<$ l'opérateur de précédence, $D_{i,j}$ le nombre de données à transmettre de la tâche i à la tâche j, A_i le nombre d'instructions à exécuter par la tâche i et W_i le cout de la tâche i.



P1	P2	P3
10	9	8
7	6	5
4	3	2
		1

FIG. 7.3 – *Niveau des tâches*

En général, le problème d'ordonnancement statique est NP-complet (problème de minimisation sous contraintes). Mais il existe des méthodes simples dans certains cas.

7.2.1 Premier algorithme simple

Le critère que l'on choisit pour l'ordonnancement des tâches est de minimiser le maximum des temps auxquels les processeurs n'ont plus rien à exécuter. Si l'on suppose que toutes les tâches ont le même temps d'exécution, et que le graphe des tâches est un arbre fini, alors l'algorithme présenté (l'algorithme de Hu) dans cette section est optimal.

On appelle niveau d'un noeud x dans l'arbre des tâches le nombre maximum de noeuds (incluant x) des chemins de x vers une feuille quelconque.

On appelle priorité d'une tâche son niveau. Ainsi, quand un processus se libère, on lui donne la tâche avec priorité maximale. La figure 7.3 est un exemple.

7.2.2 Deuxième algorithme

On suppose maintenant que le graphe de tâches est un graphe acyclique quelconque mais on se restreint au cas où l'on a $m = 2$ processeurs.

Le but est de retrouver une notion de niveau l,

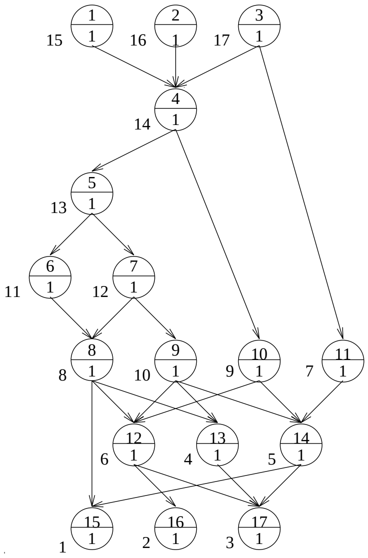
- On pose $l(x) = 1$ pour toutes les feuilles x,
- On suppose que l'on a utilisé les niveaux 1 à $j-1$. Soit S l'ensemble des tâches dont on n'a pas le niveau et dont tous les successeurs ont un niveau.
- Pour tout x de S on pose $k(x) = (l(y_1), \cdots, l(y_u))$, la suite des niveaux des successeurs de x écrits dans l'ordre décroissant.
- Soit x dans S avec $k(x)$ minimal (ordre lexicographique) dans $k(S)$. Alors on pose $l(x) = j$.

La figure 7.4 fournit un exemple d'utilisation de cet algorithme.

Dans la pratique, après avoir imaginé un algorithme, il faut trouver les tâches (ici sans boucles...), examiner le problème du placement des tâches, et en déduire les "spawn" et les "barrier" nécessaires.

Jusqu'à présent nous n'avons pas mis en oeuvre les communications. Cela complique encore le problème.

Il existe un certain nombre d'heuristiques possibles. On peut changer le "grain" de parallélisme: tâches plus grosses, moins de délai à cause des communications



P1	P2
3	2
1	11
4	
5	10
7	6
9	8
12	14
13	16
17	15

FIG. 7.4 – *Exemple simple d'utilisation du deuxième algorithme d'ordonnancement*

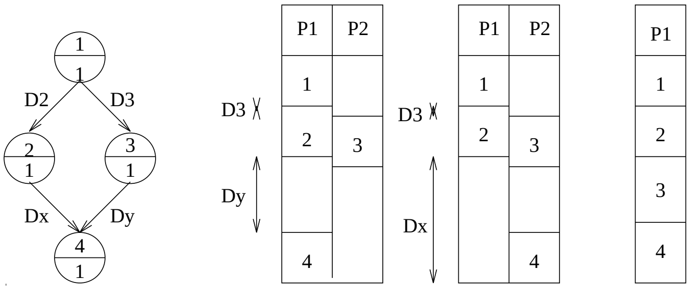


FIG. 7.5 – *Ordonnancement en présence de communications*

mais moins de parallélisme. Ou encore on peut utiliser les méthodes précédentes mais avec une notion différente de niveau. Enfin, dans le cas où les communications sont très chères on peut même en arriver à faire de la duplication de tâches!

Pour considérer les communications, on inclut les "tâches de communication", donc le niveau de x est le maximum des temps d'exécution et de communication des chemins partant de x. Le problème est que le cout de communication dépend de l'allocation des tâches en général! Un algorithme possible est le suivant: on commence par choisir un ordonnancement, puis on calcule les niveaux en incluant les couts de communication. Enfin on itère ce procédé jusquà obtenir une solution stable.

Ensuite on doit faire une analyse de performances plus fine. Eventuellement on peut remplacer des `pvm_recv` bloquants par des `pvm_nrecv` non-bloquants et améliorer ainsi le recouvrement calculs/communications.

7.2.3 Ordonnancement de boucles

On considère plus spécifiquement ce qui se passe dans des boucles imbriquées telles que,

```
for (i1=l1;i1<=m1;i1++)
 for (i2=l2;i2<=m2;i2++)
  etc.
   for (ik=lk;ik<=mk;ik++)
    Corps;
```

On appelle espace d'itération l'ensemble de valeurs que peut prendre le vecteur $(i1, i2, \cdots, ik)$ (sous-espace de $\mathbb{N}^k$).

Parallélisation par analyse de dépendance

La littérature distingue les notions de dépendance de flot (δ^f), anti (δ^a), et de sortie (δ^o). On a dépendance de flot entre une ligne de code l et une autre m, accessible depuis l, si l assigne une valeur à une variable qui est lue en m. On a parcontre dépendance anti si on lit en l une variable qui est affectée en m. Enfin, la dépendance de sortie est le cas où on lit en l et en m la même variable. La notion importante pour la parallélisation est celle de dépendance de flot: on ne peut exécuter en parallèle sans synchronisation deux lignes ou deux tâches avec une dépendance de flot.

Les graphes de dépendances sont des graphes indiquant la dépendance d'une itération $(i1, \cdots, ik)$ à une itération $(j1, \cdots, jk)$.

Par exemple, pour le code suivant, on a le graphe de dépendance de la figure 7.6:

```
S1: x=y+1;
for (i=2;i<=30;i++)
 {
S2: C[i]=x+B[i];
S3: A[i]=C[i-1]+z;
S4: C[i+1]=B[i]*A[i];
for (j=2;j<=50;j++)
 {
S5:  F[i,j]=F[i,j-1]+x;
 }
}
S6: z=y+3;
```

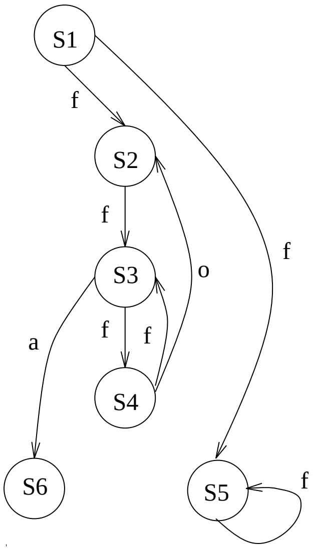


FIG. 7.6 – *Graphe de dépendance*

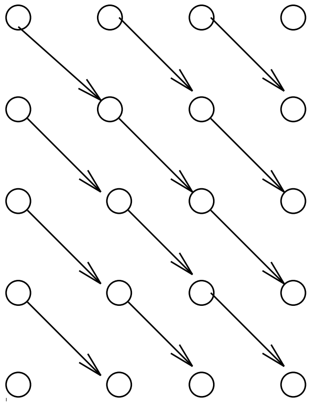


FIG. 7.7 – *Un autre graphe de dépendance*

Ils permettent de trouver les calculs suffisament indépendants pour être parallélisés.

Par exemple on considère le code suivant qui a pour graphe de dépendance le graphe de la figure 7.7:

```
for (i=1;i<=5;i++)
 for (j=1;j<=4;j++)
  X[i+1,j+1]=X[i,j]+Y[i,j];
```

Dans le cas où, comme dans ce code, le corps des boucles est composé de calculs sur des tableaux (c'est ce qui implique les dépendances), et l'accès à un indice de tableau dans les boucles se fait par un calcul linéaire sur le vecteur d'itération $(i1, \cdots, ik)$ alors le calcul de dépendance peut se faire en résolvant des systèmes d'équations Diophantiennes. C'est le cas typiquement de schémas de calcul numérique.

Prenons l'exemple suivant:

```
for (i=1;i<=101;i++)
```

```
 {
S1: A[2*i]=...;
S2: ...=A[3*i+198];
 }
```

Pour savoir si on peut avoir dépendance de flot entre S1 et S2, il faut trouver des solutions à $2x = 3y + 198$ avec $1 \leq x, y \leq 101$. La solution générale est $x = 3t + 396$, $y = 2t + 198$. D'où $-131 \leq t \leq -99$ et $-98 \leq t \leq -49$, ce qui est impossible. Donc il n'y a pas de dépendance et les boucles sur S1 et sur S2 peuvent être faites sur deux processeurs différents.

7.2.4 Optimisation de la parallélisation

L'optimisation par transformation du code source peut se faire étant donnée l'analyse de dépendance précédente. Pour cela il y a un certain nombre de transformations élémentaires que l'on peut effectuer, et que certains compilateurs parallélisants peuvent faire de façon automatique.

Fusion de boucles

Par exemple:

```
forall (i=1;i<=N;i++)
 D[i]=E[i]+F[i];
forall (j=1;j<=N;j++)
 E[j]=D[j]*F[j];
```

devient,

```
forall (i=1;i<=N;i++)
 {
  D[i]=E[i]+F[i];
  E[j]=D[j]*F[j];
 }
```

Cela permet une vectorisation et donc une réduction du cout des boucles parallèles.

Composition de boucles

Le code:

```
forall (j=1;j<=N;j++)
 forall (k=1;k<=N;k++)
  ...
```

devient,

```
forall (i=1;i<=N*N;i++)
 ...
```

Cela permet de changer l'espace d'itérations (afin d'effectuer éventuellement d'autres transformations).

Distribution de boucles

Le code:

```
for (i=1;i<=N;i++)
 { A[i+1]=B[i-1]+C[i];
   B[i]=A[i]*K;
   C[i]=B[i]-1; }
```

devient,

```
for (i=1;i<=N;i++)
 { A[i+1]=B[i-1]+C[i];
   B[i]=A[i]*K; }
forall (i=1;i<=N;i++)
   C[i]=B[i]-1;
```

Echange de boucles

Le code:

```
forall (i=2;i<=N;i++)
 for (j=2;j<=M;j++)
  A[i,j]=A[i,j-1]+1;
```

devient,

```
for (j=1;j<=M;j++)
 A[1:N,J]=A[1:N,j-1]+1;
```

Division de noeud

Le code:

```
for (i=1;i<=N;i++)
 { A[i]=B[i]+C[i];
   D[i]=A[i-1]*A[i+1]; }
```

devient,

```
for (i=1;i<=N;i++)
 { A[i]=B[i]+C[i];
   temp[i]=A[i+1];
   D[i]=A[i-1]*temp[i]; }
```

Le cycle interne de dépendance est ainsi remplacé par une anti-dépendance. Donc après distribution de boucle, on aura:

```
forall (i=1;i<=N;i++)
 temp[i]=A[i+1];
forall (i=1;i<=N;i++)
 A[i]=B[i]+C[i];
forall (i=1;i<=N;i++)
 D[i]=A[i-1]*temp[i];
```

Réduction de boucle

Dans le cas où le cycle interne de dépendance est de type dépendance de flot on ne peut pas utiliser la méthode précédente mais, par exemple:

```
for (i=3;i<=N;i++)
 { A[i]=B[i-2]-1;
   B[i]=A[i-3]*k; }
```

devient,

```
for (j=3;j<=N;j=j+2)
 forall (i=j; i<=j+1; i++)
  { A[i]=B[i-2]-1;
    B[i]=A[i-3]*k; }
```

Déroulement de boucle

Le code:

```
for (i=1;i<=100;i++)
 A[i]=B[i+2]*C[i-1];
```

devient,

```
for (i=1;i<=99;i=i+2)
 {
  A[i]=B[i+2]*C[i-1];
  A[i+1]=B[i+3]*C[i];
 }
```

Rotation de boucle

Le code:

```
for (i=2;i<=N-1;i++)
 for (j=2;j<=N-1;j++)
  A[i,j]=(A[i+1,j]+A[i-1,j]+A[i,j+1]+A[i,j-1])/4;
```

Il y a un front d'onde, cela devient,

```
forall (i=2;i<=N-1;i++)
 for (j=i+2;j<=i+N-1;j++)
  A[i,j-1]=(A[i+1,j-i]+A[i-1,j-i]+A[i,j+1-i]+A[i,j-1-i])/4;
```

7.3 Un exemple complet de parallélisation de code

Ce qui va nous intéresser ici est la phase de *remontée* qui est la dernière étape de la résolution du système linéaire (après une triangularisation de Gauss ou une décomposition de Cholesky).

Soit donc à résoudre le système triangulaire supérieur

$$Ux = b$$

avec,

$$U = \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} & \cdots & U_{1,n} \\ 0 & U_{2,2} & U_{2,3} & \cdots & U_{2,n} \\ & & & \cdots & \\ 0 & 0 & \cdots & 0 & U_{n,n} \end{pmatrix}$$

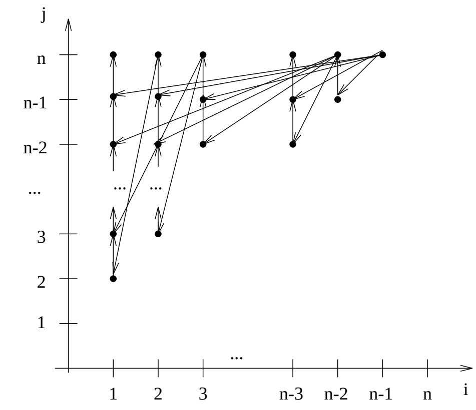


FIG. 7.8 – *Graphe de dépendance pour $n-1$ processeurs en mémoire partagée*

et $U_{i,i} \neq 0$ pour tout $1 \leq i \leq n$.

On procède par "remontée" c'est à dire que l'on calcule successivement,

$$\begin{aligned} x_n &= \frac{b_n}{U_{n,n}} \\ x_i &= \frac{b_i - \sum_{j=i+1}^{n} U_{i,j} x_j}{U_{i,i}} \end{aligned}$$

pour $i = n-1, n-2, \cdots, 1$.

Le code séquentiel correspondant est alors:

```
x[n]=b[n]/U[n,n];
for (i=n-1;i>=1;i--)
{
x[i]=0;
for (j=i+1;j<=n;j++)
      L: x[i]=x[i]+U[i,j]*x[j];
      x[i]=(b[i]-x[i])/U[i,i];
      }
```

On a représenté dans la figure 7.8 le graphe de dépendances des tâches pour $n-1$ processeurs en mémoire partagée.

Il s'agit essentiellement de dépendances de flot. On a aussi des anti-dépendances (qui n'ont absolument aucune importance en ce qui concerne la parallélisation et que l'on n'a donc pas représenté sur la figure) des itérations (i, j) vers $(i-1, j)$ (et ainsi de suite par transitivité).

7.3.1 Amélioration en mémoire partagée

En faisant une rotation et une distribution de boucles et en rajoutant des étiquettes pour faciliter la discussion sur la complexité on obtient le code suivant:

```
H': forall (i=1;i<=n-1;i++)
       x[i]=b[i];
T': x[n]=b[n]/U[n][n];
H:  for (t=1;t<=n-1;t++)
     forall (i=1;i<=n-t;i++)
      L: x[i]=x[i]-x[n-t+1]*U[i][n-t+1];
     T: x[n-t]=x[n-t]/U[n-t][n-t];
```

Si on ne regarde que la partie H du code (la seule vraiment importante asymptotiquement), on a un coût de 2 pour `L`, et un coût de 1 pour rapatrier `x[n-t+1]` Tous ces calculs se recouvrent dans le `forall`. Ils sont donc à sommer pour chaque boucle sur `t`, tout comme le coût égal à un de `T`, donc un coût de $4(n-1)$ pour `H`. A cela, il faut ajouter le temps pour `H'` égal à un et encore un pour `T'`. Donc un coût total de $4n-2$. Pour le code séquentiel, si on compte l'affectation à zéro comme une opération élémentaire de coût unitaire, le coût total est,

$$1+3(n-1)+2(1+2+\cdots+n-1)=(n-1)(n+3)+1$$

Le ratio d'accélération est donc d'ordre $\frac{n}{4}$ asymptotiquement.

7.3.2 Passage de messages

On rend explicite les communications entre les processeurs qui gèrent chacun une partie du tableau `U`. De fait, on suppose que l'on a n processeurs $P_1, \cdots, P_n$ et que l'on répartit les tableaux de la façon suivante,

P_1	P_2	P_3	$\cdots$	P_{n-1}	P_n
$b[1]$	$b[2]$	$b[3]$	$\cdots$	$b[n-1]$	$b[n]$
$U[1,n]$ $U[1,n-1]$ $\cdots$ $U[1,1]$	$U[2,n]$ $U[2,n-1]$ $\cdots$	$U[3,n]$ $U[3,n-1]$ $\cdots$	$\cdots$ $\cdots$ $\cdots$	$U[n-1,n]$ $U[n-1,n-1]$	$U[n,n]$

Le code PVM correspondant est comme suit. P_n est le maître qui a pour code:

```
T(n,1): x[n]=b[n]/U[n][n];
       /* tids est le tableau d'identificateurs */
       /* esclave est le code executable des autres processus */
       pvm_spawn("esclave",NULL,PvmTaskDefault,"",n-1,tids);
       for (i=1;i<=n-1;i++)
          {
             pvm_initsend(PvmDataDefault);
             pvm_pkint(&i,1,1);
             /* on envoie la colonne i */
             pvm_pkfloat(U[i],n,1);
             pvm_pkfloat(&(b[i]),1,1);
             pvm_pkfloat(&(x[n]),1,1);
             pvm_pkint(tids,n-1,1);
             /* TAG est le filtre d'envoi d'initialisation */
             pvm_send(tids[i-1],TAG);
          }
T(n,2): for (i=1;i<=n-1;i++)
           {
              /* RES est le filtre d'envoi des resultats */
              pvm_recv(tids[i-1],RES);
              pvm_upkfloat(&(x[i]),1,1);
           }
        /* le resultat est dans le vecteur x[] */
```

Les esclaves P_i $(1 \leq i \leq n-1)$ ont tous le même code "esclave" suivant (avec des annotations pour simplifier la décomposition en tâches):

```
T(i,1): pvm_recv(pvm_parent(),TAG);
```

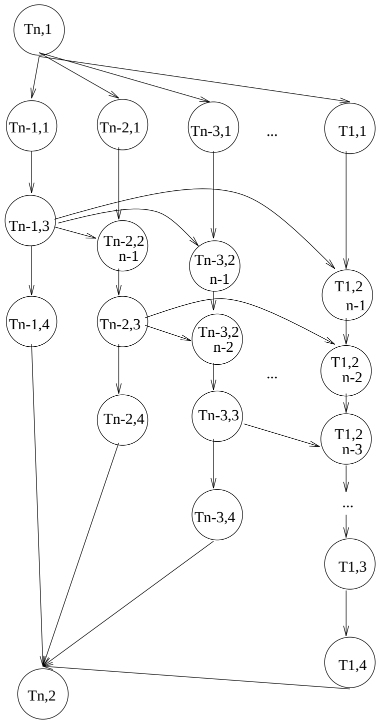


FIG. 7.9 – *Graphe de dépendance pour n processeurs distribués*

```
        pvm_upkint(&i,1,1);
        pvm_upkfloat(U[i],n,1);
        pvm_upkfloat(&(b[i]),1,1);
        pvm_upkfloat(&(x[n]),1,1);
        pvm_upkint(tids,n-1,1);
        x[i]=U[i][n]*x[n];
for (j=n-1;j>i;j--)
   {
T(i,2,j):  /* INTER est le filtre de communication entre esclaves */
           pvm_recv(tids[j],INTER);
           pvm_upkfloat(&(x[j]),1,1);
           x[i]=x[i]+U[i][j]*x[j];
   }
T(i,3): x[i]=(b[i]-x[i])/U[i][i];
        for (j=1;j<i;j++)
           {
              pvm_initsend(PvmDataDefault);
              pvm_pkfloat(&(x[i]),1,1);
              pvm_send(tid[j],INTER);
           }
T(i,4): pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&(x[i]),1,1);
        pvm_send(pvm_myparent(),RES);
```

Le graphe de dépendance est décrit dans la figure 7.9.

En supposant que le coût de l'envoi et le coût de la réception (hors attente) est égal à la taille des données envoyées multipliée par α ($0 \leq \alpha \leq 1$) et ne dépend ni du processeur ni de l'historique des calculs. En supposant également α très petit et n pas trop élevé, on a pour chaque processus des temps qui se recouvrent comme suit:

P_n	P_{n-1}	$\cdots$	P_1
$T_{n,1} : 1 + 2\alpha(n-1)(n+1)$	$--$	$\cdots$	$--$
$--$	$T_{n-1,1} : 2\alpha(n+1) + 1$	$\cdots$	$T_{1,1} : 2\alpha(n+1) + 1$
$--$	$T_{n-1,3} : 2 + \alpha(n-2)$	$\cdots$	$--$
$--$	$T_{n-1,4} : \alpha$	$\cdots$	$T_{1,2,n-1} : \alpha + 2$
$T_{n,2} : \alpha$	$--$	$\cdots$	$--$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
α	$--$	$\cdots$	$--$

Donc la première ligne du tableau correspond à un coût global de $1 + 2\alpha(n-1)(n+1)$, puis les deux premières à $2 + 2\alpha(n+1)$, puis les quatre premières: $6 + \alpha(2n(n+1)+n-1)$, puis si on somme les lignes deux par deux, on rajoute $2+\alpha(n-i)$ $(3 \leq i \leq n)$ et $\alpha + 2$. On termine enfin par ajouter α (pour $T_{1,4}$) et α (pour $T_{n,2}$). Donc le coût global est,

$$6 + 4(n-2) + \alpha(2n(n+1) + (n-1) + (n-2) + \cdots + 1) + 2\alpha$$

donc le coût est $4n - 2 + \frac{\alpha}{2}(5n^2 + 3n + 4)$.

Le coût est donc exactement celui du programme SIMD plus le coût explicite des communications. On aurait pu mieux ordonnancer les envois et gagner un peu sur ce temps supplémentaire, mais il serait resté en αn^2. C'est le coût des compactage et décompactage des données tel qu'il est programmé ici.

Une autre solution aurait été de répartir les lignes de `U` sur les processeurs modulo `p` (donc les lignes `U[1,*]`, `U[1+p,*]`, $\cdots$, `U[1+(k-1)*p,*]` sur P_1 etc.). Le nouveau code agit alors comme si on avait recopié p fois le code du maître et p fois le code des esclaves. Cela ne change rien à la complexité de l'algorithme, si on a bien fait attention à la gestion des "`broadcast`" vers les processus de numéro plus élevés.

Encore une autre solution serait de mettre des lignes adjacentes de `U` sur chacun des processeurs, c'est à dire de découper `U` en bandes horizontales. On suppose que l'on fait gérer les lignes j, $1 \leq k_i \leq j \leq l_i \leq n$ au processeur i $(1 \leq i \leq n)$, et que $i' > i$ implique $k_{i'} > l_{i'}$, plus $k_{i+1} = l_i + 1$, $k_1 = 1$, $l_p = n$. Le code PVM correspondant est comme suit.

On a le squelette suivant, pour le maître (P_p):

```
/* resolution pour les inconnues x[k[p]] a x[n] */
T(p,1):...

        /* tids est le tableau d'identificateurs */
        /* esclave est le code executable des autres processus */
        pvm_spawn("esclave",NULL,PvmTaskDefault,"",n-1,tids);
        for (i=1;i<=n-1;i++)
           {
              pvm_initsend(PvmDataDefault);
              pvm_pkint(&i,1,1);
              /* on envoie la colonne i */
              pvm_pkfloat(U[i],n,1);
              pvm_pkfloat(&(b[i]),1,1);
              pvm_pkfloat(&(x[k[p]]),n-k[p]+1,1);
              /* on envoie k_i, l_i mis dans un tableau k[] et l[] */
              pvm_pkint(k,p,1);
```

```
             pvm_pkint(l,p,1);
             pvm_pkint(tids,n-1,1);
             /* TAG est le filtre d'envoi d'initialisation */
             pvm_send(tids[i-1],TAG);
          }
T(n,2): for (i=1;i<=n-1;i++)
           {
              /* RES est le filtre d'envoi des resultats */
              pvm_recv(tids[i-1],RES);
              pvm_upkfloat(&(x[k[i]]),l[i]-k[i]+1,1);
           }
        /* le resultat est dans le vecteur x[] */
```

Les esclaves P_i $(1 \leq i \leq p-1)$ ont tous le même code "esclave" suivant:

```
T(i,1): pvm_recv(pvm_parent(),TAG);
        pvm_upkint(&i,1,1);
        pvm_upkfloat(U[i],n,1);
        pvm_upkfloat(&(b[i]),1,1);
        pvm_upkfloat(&(x[n]),1,1);
        pvm_upkint(k,p,1);
        pvm_upkint(l,p,1);
        pvm_upkint(tids,n-1,1);
/* initialisation des variables x[k[i]] a x[l[i]] */
...

for (j=p-1;j>i;j--)
   {
T(i,2,j):  /* INTER est le filtre de communication entre esclaves */
           pvm_recv(tids[j],INTER);
           pvm_upkfloat(&(x[k[j]]),l[j]-k[j]+1,1);
           ...
           /* resolution ... */
   }
T(i,3): for (j=1;j<i;j++)
           {
              pvm_initsend(PvmDataDefault);
              pvm_pkfloat(&(x[k[i]]),l[i]-k[i]+1,1);
              pvm_send(tid[j],INTER);
           }

T(i,4): pvm_initsend(PvmDataDefault);
        pvm_pkfloat(&(x[k[i]]),l[i]-k[i]+1,1);
        pvm_send(pvm_myparent(),RES);$i' > i$
```

Avec des processeurs d'égale puissance, le mieux est de découper en domaines avec un même nombre de lignes, donc $k_i = 1 + k(i-1)$ et $l_i = ki$, Une meilleure solution consiste à envoyer `x[k[i]]` sitôt calculé, aux esclaves $P_{i'}$, $i' > i$, puis `x[k[i]+1]` jusqu'à `x[l[i]]` sitôt calculés. Cela permet de recouvrir bien mieux les calculs et les communications que dans la solution précédente.

Chapitre 8

Algorithmes évolutionnaires et problèmes inverses

Ce chapitre présente succintement les algorithmes évolutionnaires [1], méthodes stochastiques d'optimisation globale qui s'inspirent (librement) de l'évolution darwinienne des populations biologiques, et dont font partie les algorithmes génétiques. Un des intérêts d'étudier ici ces algorithmes tient dans les nombreuses variantes de parallélisation possibles – sachant que le temps calcul important nécessaire à l'obtention de résultats sur des problèmes réels rend indispensable l'utilisation de ressources distribuées. Quelques techniques de parallélisation seront donc décrites dans la section 8.2. Enfin, une approche générique de l'application de ces algorithmes à la résolution de problèmes inverses sera proposée, illustrée par un problème d'identification de sous-sol recherche pétrolière.

8.1 Les algorithmes

Les phénomènes physiques ou biologiques ont été à la source de nombreux algorithmes s'en inspirant plus ou moins librement. Ainsi les réseaux de neurones artificiels s'inspirent du fonctionnement du cerveau humain, l'algorithme de recuit simulé de la thermodynamique, et les *algorithmes évolutionnaires* (AEs) (dont les plus connus sont les *algorithmes génétiques*) de l'évolution darwinienne des populations biologiques.

Avant de décrire plus avant les algorithmes évolutionnaires précisons toutefois que le paradigme darwinien qui sous-tend ces algorithmes est une grossière caricature du phénomène naturel qu'il imite, et qu'en tout état de cause, il ne saurait être une justification de l'emploi de ces techniques – pas plus que le vol des oiseaux ne peut justifier l'invention de l'avion! [2].

8.1.1 Définitions et notations

Soit à optimiser une fonction $\mathcal{F}$ à valeurs réelles definie sur un espace Ω, supposé muni d'une distance d. Le parallèle avec l'évolution naturelle a entrainé l'apparition d'un vocabulaire spécifique :

- La fonction objectif $\mathcal{F}$ est appelée fonction *performance*, ou fonction *d'adaptation* (*fitness* en anglais) ;

1. On n'hésitera pas à consulter les livres ou articles [BFM97], [BNKF97], [SAB98] et [Mic96].
2. on peut même utiliser ce type d'algorithme en n'étant pas un convaincu du darwinisme, d'o d'ailleurs l'emploi du néologisme "évolutionnaire", inspiré de l'anglais *evolutionary*.

– Les points de l'espace de recherche Ω sont appelés des *individus* ;
– Les P-uplets d'individus sont appelés des *populations* ;
– On parlera d'une *génération* pour la boucle principale de l'algorithme ;

8.1.2 Le squelette

La pression de l'"environement" est simulée à l'aide de la fonction d'adaptation $\mathcal{F}$, et le principe darwinien *Les plus adaptés survivent et se reproduisent*, est implanté de la manière suivante :

– **Initialisation** de la *population* Π_0 en choisissant P individus dans Ω, généralement par tirage aléatoire avec une probabilité uniforme sur Ω ;
– **Evaluation** des individus de Π_0 (i.e. calcul des valeurs de $\mathcal{F}$ pour tous les individus) ;
– La génération i construit la population Π_i à partir de la population Π_{i-1} :
  – **Sélection** des individus les plus performants (au sens de $\mathcal{F}$) de Π_{i-1} ;
  – Application (avec une probabilité donnée) des **opérateurs génétiques** aux *parents* sélectionnés, ce qui génère de nouveaux individus, les *enfants* ; on parlera de *mutation* pour les opérateurs unaires, et de *croisement* pour les opérateurs binaires (ou n-aires) ;
  – **Evaluation** des enfants ;
  – **Remplacement** des parents au moyen d'une sélection darwinienne parmi les enfants, avec participation éventuelle des parents.
– L'évolution stoppe quand le niveau de performance souhaité est atteint, ou qu'un nombre fixé de générations s'est écoulé sans améliorer l'individu le plus performant.

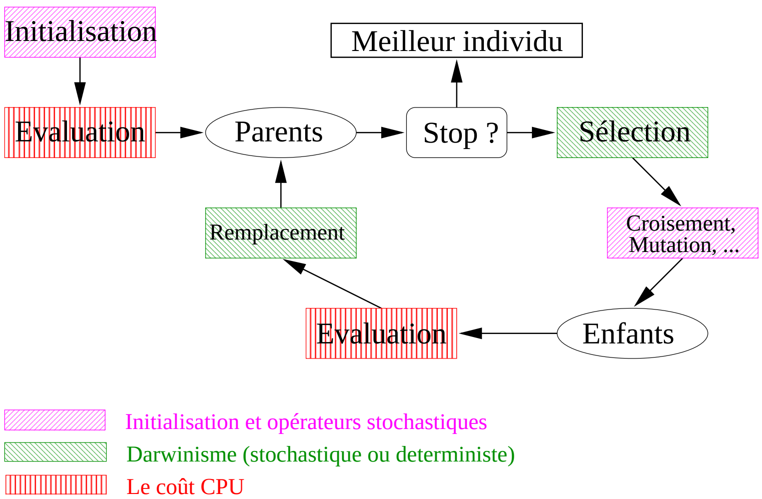


FIG. 8.1 – Squelette d'un algorithme évolutionaire

Il est important de noter que le coût CPU essentiel de ces algorithmes provient de l'étape d'évaluation (calcul des performances) : les tailles de populations sont de l'ordre de quelques dizaines, le nombre de générations de quelques centaines, ce qui donne lieu à quelques dizaines de milliers de calculs de $\mathcal{F}$.

8.1.3 Les composants

Nous allons maintenant détailler les principaux composants d'un algorithme évolutionnaire, en donnant des exemples concrets.

On notera tout d'abord que l'on peut répartir les diverses étapes de l'algorithme de la section précédente en deux groupes : celles qui dépendent de l'espace de recherche (initialisation, opérateurs génétiques et évaluation) et celles qui n'en dépendent pas (sélection et remplacement).

L'étape d'évaluation, totalement spécifique au problème traité, et objet des sections sur la parallélistion, ne sera plus mentionnée ici.

L'espaces de recherche

Il s'agit de la composante principale de l'algorithme – qui est même en fait préalable aux autres. Dans de nombreux cas, l'espace de recherche est totalement déterminé par le problème (la fonction objectif). Mais il est toujours possible de transporter son problème dans un espace habilement choisi (changement de "variables") où sa résolution sera plus aisée. Cet espace, où seront appliqués les opérateurs génétiques, est alors appelé *espace génotypique*, et l'espace de recherche initial, dans lequel est calculée la performance des individus, est appelé *espace phénotypique*.

Nous allons maintenant donner deux exemples d'espaces de recherche parmi les plus utilisés – et détaillerons les composantes de l'algorithmes dans ces deux cas particulier.

LES CHAÎNES DE BITS (*bitstring* en anglais).
L'espace de recherche est ici $\Omega = \{0,1\}^N$. Historiquement (voir section 8.1.4) il s'agit de l'école des algorithmes génétiques, et la jsutification de l'utilisation intensive de cet espace de recherche particulier était fondé à la fois sur un parallèle encore plus precis avec la biologie (une chaîne de bits étant assimilée à un chromosome) et sur des considérations théoriques qui ne seront pas détaillées ici (voir les références de la section 8.1.4 à ce sujet). Ce contexte reste toutefois très utilisé, et permet également une présentation aisée des divers composants de l'algorithme.

LES VECTEURS DE RÉELS:
C'est bien sûr le cas le plus fréquent en calcul numérique : Ω est un sous-ensemble de $\mathbb{R}^n$, borné ou non. On parle alors aussi d'*optimisation paramétrique*.

Initialisation

Le principe général de l'initialisation est d'échantilloner le plus uniformément possible l'espace de recherche Ω.

Dans le cas des chaînes de bits, chaque bit de chaque individu est tiré égal à 0 ou à 1 avec un probabilité 0.5.

Dans le cas de l'optimisation paramétrique, si $\Omega = \prod[a_i, b_i]$ (cas borné), on tire uniformément chaque coordonnée dans l'intervalle correspondant. Par contre, si Ω n'est pas borné, il faut faire des choix. On pourra par exemple tirer mantisses et exposants uniformément.

Opérateurs génétiques

C'est durant cette étape que de nouveaux individus sont crées à partir des individus préalablement sélectionnés. On distingue les opérateurs de croisement (binaires, ou plus généralement n-aires) et les opérateurs de mutation, unaires. A noter que cette étape est toujours **stochastique**, c'est à dire que le résultat de l'application d'un opérateur dépend de tirages aléatoires.

LE CROISEMENT
L'idée générale du croisement est l'échange de *matériel génétique* entre les *parents*: si deux parents sont plus performants que la moyenne, on peut espérer que cela est du à certaines parties de leur *génotype*, et que certains des enfants, recevant les"bonnes" parties de leurs deux parents, n'en seront que plus performants. Ce raisonnement, trivialement valable pour des fonctions performance linéaires, est extrapolé (et expérimentalement vérifié) à une classe plus étendue de fonctions, sans que les résultats théoriques aujourd'hui disponibles ne permettent de délimiter précisément la classe de fonctions pour lesquelles le croisement est utile.

Signalons enfin que tous les individus sélectionnés ne sont pas forcément croisés. Le croisement – comme tous les opérateurs génétiques, est appliqué avec une certaine probabilité notée P_c.

Dans le cadre des **chaînes de bits**, les divers opérateurs de croisement échangent des bits (à position fixée) entre les parents. La figure 8.2 donne l'exemple du croisement à point. Un autre croisement consiste à tirer au sort pour chaque position (avec probabilité 0.5) de quel parent proviendra le bit correspondant chez chaque enfant.

$$\left.\begin{matrix}(b_1,\ldots,b_N)\\(c_1,\ldots,c_N)\end{matrix}\right\} \xrightarrow{P_c} \left\{\begin{matrix}(b_1,\ldots,b_l,c_{l+1},\ldots,c_N)\\(c_1,\ldots,c_l,b_{l+1},\ldots,b_N)\end{matrix}\right.$$

FIG. 8.2 – *Croisement à un point: l'entier l est tiré uniformément dans $[1, N-1]$, et les deux moitiés des chromosomes sont échangées.*

Dans le cadre de l'**optimisation paramétrique**, on peut bien entendu appliquer les mêmes opérateurs d'échange de coordonnées. On peut également – et c'est en général bien plus efficace – "mélanger" les deux parents par une combinaison linéaire. On parle alors de *croisement arithmétique*:

$$(\boldsymbol{X}, \boldsymbol{Y}) \xrightarrow{p_c} \alpha\boldsymbol{X} + (1-\alpha)\boldsymbol{Y}, \alpha = U[0,1]$$

ou

$$(\boldsymbol{X}, \boldsymbol{Y}) \xrightarrow{p_c} \alpha_i\boldsymbol{X_i} + \boldsymbol{(1-\alpha_i)}\boldsymbol{Y_i}, \alpha_i = U[0,1]$$

La première version revient à choisir l'enfant uniformément sur le segment $[\boldsymbol{XY}]$ alors que la deuxième revient à tirer l'enfant uniformément sur l'hypercube dont $[\boldsymbol{XY}]$ est une diagonale. Remarquons que l'échange de coordonnées revient à choisir comme enfant un des sommets de cet hypercube.

LA MUTATION L'idée directrice de la mutation est de permettre de visiter tout l'espace. Les quelques résultats théoriques de convergence des algorithmes évolutionnaires ont d'ailleurs tous comme condition l'**ergodicité** de la mutation, c'est-à-dire le fait que tout point de l'espace de recherche peut être atteint en un nombre fini de mutations.

Dans le cadre des **chaînes de bits**, on modifie aléatoirement certains bits. Par exemple, pour chaque parent (ou enfant du croisement), et pour chaque position l,

$$(b_1, b_2, \ldots, b_N) \xrightarrow{p_m} (b_1, b_2, \ldots, 1-b_l, b_{l+1}, \ldots, b_N)$$

Attention, la probabilité p_m est ici une probabilité **par bit** et non par individu.

Une autre possibilité est de prédéfinir un nombre de bits à modifier (généralement 1) et de choisir ensuite au hasard une position dans chaque individu et de modifier le bit correspondant. Cet opérateur est lui appliqué avec un probabilité P_m **par individu**, bien sûr.

Dans le cadre de l'**optimisation paramétrique**, la mutation la plus employée est la mutation *gaussienne*, qui consiste à rajouter un bruit gaussien à chaque variable (l'opérateur étant appliqué avec une certaine probabilité par individu P_m).

$$X_i := X_i + N(0, \sigma^2)$$

Tout l'art est alors dans le choix de la variance σ du bruit gaussien utilisé. On peut évidemment demander à l'utilisateur de fixer cette valeur. Mais des études théoriques sur des fonctions simples ont démontré que cette valeur devrait décroitre au fil des générations – et il devient difficile de fixer une suite de valeurs décroissante qui soit synchrone avec l'éventuelle convergence de l'algorithme pour des fonctions quelconques.

L'idée originale pour résoudre ce problème est due aux pères des stratégies d'évolution (I. Rechenberg et H.-P. Schwefel, voir section 8.1.4). Elle consiste à considérer ces σ comme des variables supplémentaires, et à les faire également évoluer (via croisement et mutation!).

Si l'espace phénotypique initial est, par exemple, $\prod[a_i, b_i]$, l'espace génotypique sera $\prod[a_i, b_i] \times \mathbb{R}^+$, chaque individu se voyant muni de sa propre variance σ. La mutation devient alors

$$\begin{aligned} \sigma &:= \sigma\ exp(\tau N(0,1)) \\ X_i &:= X_i + N(0, \sigma) \end{aligned}$$

La variance σ est modifiée par une perturbation log-normale – elle doit rester positive, et son évolution doit être symétrique par rapport à un puisqu'elle intervient ensuite de manière multiplicative.

L'idée est qu'un individu ayant de bonnes performances après plusieurs mutations n'est pas arrivé où il est par hasard, et doit avoir une valeur de σ **adaptée** à la zone de l'espace de recherche où il se trouve. Signalons qu'il existe une variante dans laquelle une variance est attachée à chaque variable. Enfin, dans la variante la plus générale, une matrice de corrélation complète évolue avec chaque individu.

Procédures de sélection

La partie darwinienne de l'algorithme comprend les deux étapes de **sélection** et de **remplacement**. Répétons que ces étapes sont totalement indépendantes de l'espace de recherche.

On distingue deux catégories de procédures de sélection ou de remplacement (par abus de langage, nous appellerons sélection les deux types de procédures) : les procédures déterministes et les procédures stochatiques.

Sélection déterministes

On sélectionne les meilleurs individus (au sens de la fonction performance). Si plus de quelques individus doivent être selectionnés, cela suppose un tri de l'ensemble de la population – mais cela ne pose un problème de temps calcul que pour des très grosses tailles de population.

Les individus les moins performants sont totalement éliminés de la population, et le meilleur individu est toujours sélectionné – on parle alors d'*élitisme*.

Sélection stochastiques
Il s'agit toujours de favoriser les meilleurs individus, mais ici de manière stochastique, ce qui laisse une chance aux individus moins performants. Par contre, il peut arriver que le meilleur individu ne soit pas sélectionné, et qu'aucun des enfants n'atteigne un performance aussi bonne que celle du meilleur parent ...

Le **tirage de roulette** est la plus célèbre des sélections stochastiques. Supposant un problème de maximisation avec uniquement des performances positives, elle consiste à donner à chaque individu une probabilité d'être sélectionné proportionnelle à sa performance. Une illustration de la roulette est donnée figure 8.3 : on lance la boule dans la roulette, et on choisit l'individu dans le secteur duquel la boule a fini sa course.

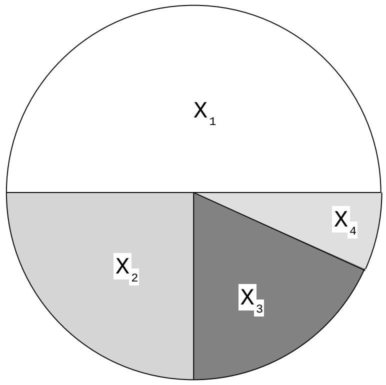


FIG. 8.3 – *Sélection par tirage de roulette: cas de 4 individus de performances respectives 50, 25, 15 et 10. Une boule est "lancée" dans cette roulette, et l'individu dnas le secteur duquel elle s'arrête est sélectionné.*

Le tirage de roulette présente toutefois de nombreux inconvénients, en particulier reliés à l'échelle de la fonction performance : alors qu'il est théoriquement équivalent d'optimiser $\alpha\mathcal{F} + \beta$ et $\mathcal{F}$ pour tout $\alpha > 0$, il est clair que le comportement de la sélection par roulette va fortement dépendre de α dans ce cas. C'est pourquoi cette sélection est presque totalement abandonnée aujourd'hui.

La **sélection par le rang** consiste à faire une sélection en utilisant une roulette dont les secteurs sont proportionnels aux rangs des individus. La variante linéaire utilise directement le rang (P pour le meilleur, 1 pour le moins bon, pour une population de taille P), les variantes polynomiales remplaçant ces valeurs par $\frac{i}{P}^{\alpha}$, $\alpha > 0$. Seuls comptent alors les position srelatives des individus entre eux, et non les valeurs – arbitraires – de la fonction $\mathcal{F}$.

La **sélection par tournoi** n'utilise aussi que des comparaisons entre individus – et ne nécessite même pas de tri de la population. Elle possède un paramètre T, taille du tournoi. Pour sélectionner un individu, on en tire T uniformément dans la population, et on sélectionne le meilleur de ces T individus. Le choix de T permet de faire varier la *pression sélective*, c'est-à-dire les chances de sélection des plus performants par rapport aux plus faibles. A noter que le cas $T = 2$ correspond, en espérance et au premier ordre en fonction de P, à la selection par le rang linéaire.

Les schémas d'évolution

On regroupe sous ce nom les ensemble sélection/remplacement, qui ne peuvent être dissociées lors des analyses du darwinisme au sein des algorithmes évolutionnaires. Un schéma d'évolution est donc la réunion d'une procédure de sélection et d'une procédure de remplacement. Toute combinaison des procédures présentées plus haut (et de bien d'autres encore) est licite. Toutefois, certaines combinaisons sont plus souvent utilisées, que ce soit pour des raisons historiques, théoriques ou expérimentales. Pour cette raison, les noms donnés sont souvent les noms des école historiques qui les ont popularisées – mais gardons à l'esprit que ces schémas sont totalement indépendants de l'espace de recherche, alors que nous verrons que les écoles historiques travaillaient sur des esapces de recherche bien précis.

SCHÉMA ALGORITHME GÉNÉTIQUE GÉNÉRATIONEL (*GGA*)
Ce schéma utilise une sélection stochastique pour sélectionner exactement P parents (certains parents peuvent donc être sélectionnés plusieurs fois, d'autres pas du tout). Ces P parents donnent ensuite P enfants par application des opérateurs génétiques (avec probabilité donnée). Enfin, ces P enfants remplacent purement et simplement les P parents pour la génération suivante.

SCHÉMA ALGORITHME GÉNÉTIQUE STATIONNAIRE (*Steady-state GA – SSGA*)
Dans ce schéma, un individu est sélectionné, généralement par tournoi, un second si le croisement doit être appliqué, et l'enfant résultant (après croisement et mutation éventuels) est re-inséré dans la population en remplacement d'un "vieux" parent sélectionné par un tournoi inversé (le moins performant "gagne").

SCHÉMAS STRATÉGIES D'ÉVOLUTION ($(\mu \overset{+}{,} \lambda)$-*ES*)
Deux schémas sont regroupés sous ces appellations. Dans les deux cas, l'étape de sélection est un tirage uniforme (on peut dire qu'il n'y a pas de sélection au sens darwinien). À partir d'une population de taille μ (notations historiques!), λ enfants sont générés par application des opérateurs génétiques. L'étape de remplacement est alors totalement déterministe. Dans le schéma (μ, λ)-ES, les meilleurs μ enfants deviennent les parents de la génération suivante, alors que dans le schéma $(\mu + \lambda)$-ES, les meilleurs des $\mu + \lambda$ parents plus enfants sont les parents de la génération suivante.

8.1.4 Les algorithmes historiques

On distingue quatre grandes familles historiques d'algorithme – et les différences entre elles ont laissé des traces dans le paysage évolutionnaire actuel, en dépit d'une unification de nombreux concepts.

- **Algorithmes Génétiques** (*GA*), J. Holland, 1975 et D.E. Goldberg, 1989, Michigan, USA.
  Les GA ont été imaginés comme outils de modélisation de l'adaptation. Ils travaillent dans l'espace des chaînes de bits $[0, 1]^n$ avec les schémas GGA et SSGA. Ce sont les plus connus des algorithmes évolutionnaires, et (malheureusement ?) souvent les seuls variantes connues des chercheurs des autres disciplines.
- **Stratégies d'évolution** (*ES*), I. Rechenberg et H.P. Schwefel, 1965, Berlin.
  Les ES ont été mises au points par deux jeunes ingénieurs travaillant sur des problèmes numériques. Le contexte était l'optimisation paramétrique, et les schémas d'évolution sont bien entendu les $(\mu \overset{+}{,} \lambda)$-ES. Un énorme progrès a été apporté par les techniques **adaptatives** d'ajustement des paramètres de mutation, et ce sont sans contestation les meilleurs algorithmes pour les problèmes purement numériques.
- **Programmation évolutionnaire** (*EP*), L.J. Fogel, 1966 et D.B. Fogel, 1991, 1995, Californie, USA.
  Imaginée par Larry Fogel pour la découverte d'automates à états finis pour

l'approximation de séries temporelles, EP a rapidement travaillé sur des espaces de recherche très variés. Le schéma utilisé ressemble à s'y méprendre à un $(P + P)$-ES – quoique développé complètement indépendemment.

- **Programmation génétique** (*GP*), J. Koza, 1990, Californie, USA.
  Apparue initialement comme sous-domaine des GAs, GP est devenu une branche à part entière (conférence, journal, ...). La spécificité de GP est l'espace de recherche, fait d'arbres représentant des programmes complets. GP cherche à atteindre (et réussit souvent!) un des vieux rêves des programmeurs, faire écrire le programme par un autre programme. Les schémas d'évolution utilisés sont souvent de type SSGA, mais avec des tailles de population énormes. Et les tendance récentes sont pour GP ...la parallélisation systématique et sur de grosses grappes de stations. Ainsi, les résultats récents les plus spectaculaires obtenus par Koza l'ont été avec 64 populations de 10000 individus, ou encore avec 1000 populations de 500 individus, utilisant le modèle en îlots (voir section 8.2.3) sur des grappes *Beowulf* de stations Alpha ou Pentium sous Linux.

8.1.5 Les points-clé

La diversité génétique

Le terme *diversité génétique* désigne la variété des génotypes présents dans la population. Elle devient nulle lorsque tous les individus sont identiques – on parle alors de *convergence* de l'algorithme (a posteriori!). Mais il est important de savoir que lorsque la diversité génétique devient très faible, il y a très peu de chances pour qu'elle augmente à nouveau. Et si cela se produit trop tôt, la convergence a lieu vers un optimum local – on parle alors de *convergence prématurée*. Il faut donc préserver la diversité génétique, sans pour autant empécher la convergence. Un autre point de vue sur ce problème est le suivant.

Le dilemne exploration-exploitation

A chaque étape de l'algorithme, il faut effectuer le compromis entre **explorer** l'espace de recherche, afin d'éviter de stagner dans un optimum local, et **exploiter** les meilleurs individus obtenus, afin d'atteindre de meilleurs valeurs aux alentours. Trop d'exploitation entraîne une convergence vers un optimum local, alors que trop d'exploration entraîne la non-convergence de l'algorithme.

Typiquement, les opérations de sélection et de croisement sont des étapes d'exploitation, alors que l'initialisation et la mutation sont des étapes d'exploitation. On peut ainsi régler les parts respectives d'exploration et d'exploitation en jouant sur les divers paramètres de l'algorithme (probabilités d'application des opérateurs, pression de sélection, ...). Malheureusement, il n'existe pas de règles universelles de réglages et seuls des résultats expérimentaux donnent une idée du comportement des divers composantes des algorithmes.

8.2 Parallélisation des AEs

Les algorithmes évolutionnaires sont des algorithmes coûteux en temps calcul – mais il est facile de voir que l'étape la plus coûteuse – l'évaluation de la performance de toute une population – est constituée de calculs totalement indépendants entre eux. Il semble donc facile de les paralléliser.

Cependant, la méthode naturelle de lancement en parallèle des calculs de performance n'est pas la seule manière d'envisager la parallélisation des algorithmes évolutionnaires. Il existe même une gamme complète de manière de paralléliser ces algorithmes, de la simple parallélisation du calcul de performance lui-même jusqu'à la distribution complète de la population sur les divers processeurs disponibles.

Une quantité clé pour le choix de la méthode de parallélisation comme pour le calcul du gain que celle-ci apporte est le *ratio évaluation-évolution*, rapport entre le temps moyen pour un calcul de performance et le temps moyen nécessaire pour la sélection, l'application des opérateurs génétiques et le remplacement d'un individu (temps moyen, car ces opérations sont généralement effectuées sur des groupes d'individus).

8.2.1 Parallélisation du calcul de performance

Cette technique est citée pour mémoire, car rien de général ne peut être dit a ce sujet : si l'on dispose déjà d'un algorithme de calcul de la fonction performance parallèle, on peut bien sûr l'utiliser au sein d'un algorithme évolutionnaire standard. Le gain espéré dépend alors totalement du problème.

8.2.2 Distribution du calcul de performance

Distribution synchrone

Il s'agit donc de paralléliser l'étape d'**évaluation**. Une station maître gère l'algorithme lui-même (sélection/remplacement et opérateurs génétiques), et envoie les calculs de performance à des stations esclaves. Le comportement de l'algorithme parallèle est alors exactement celui de l'algorithme séquentiel, et le facteur d'accélération devient rapidement très proche du nombre de processeurs lorsque le ratio évaluation-évolution augmente, pour des tailles de population ne dépassant pas quelques multiples du nombre de processeurs disponibles.

Lorsque la durée d'un calcul de performance varie d'un calcul à l'autre (que ce soit en raison de conditions de calcul différentes, ou de par l'hétérogénéité des processeurs utilisés), le gain apporté par ce schéma de parallélisation peut toutefois se dégrader, l'ensemble des processeurs devant alors attendre le plus lent d'entre eux. Une gestion plus fine de la distribution des calculs (à l'aide d'une liste de calculs à effectuer, un processeur recevant un nouveau calcul dés la fin du calcul précédent) permet de limiter ce phénomène sans complètement le suprimer en fin de génération. De plus, la tolérance aux pannes est nulle : si un processeur s'arrête pour une raison quelconque, l'ensemble de l'algorithme se mettra en attente du résultat manquant.

Distribution asynchrone

Une manière élégante de régler ces problèmes est d'utiliser un schéma d'évolution asynchrone : chaque processeur esclave effectue son calcul de performance, renvoie le résultat au maître et reçoit immédiatement un nouveau calcul. Cela suppose que les étapes de sélection/remplacement peuvent être effectuée individu par individu. Le schéma *steady-state* séquentiel permet déjà cela, et peut donc être utilisé sans modification : chaque fois qu'un esclave renvoie la performance d'un enfant, le maître effectue l'étape de remplacement pour cet enfant, immédiatement suivie d'une sélection de nouveaux parents et de l'appliation des opérateurs génétiques, avant de renvoyer l'enfant à évaluer à l'esclave en question, qui ainsi n'attend pas.

Un premier inconvénient est que certains individus dont le calcul de performance aura été plus long (convergence plus lente d'un algorithme numériques ou processeur moins rapide) seront insérés dans une population ayant évolué depuis leur apparition, et risquent d'être anachroniques – et en particulier de d'être éliminés rapidement, avant d'avoir été sélectionnés pour reproduction. Mais il ne semble

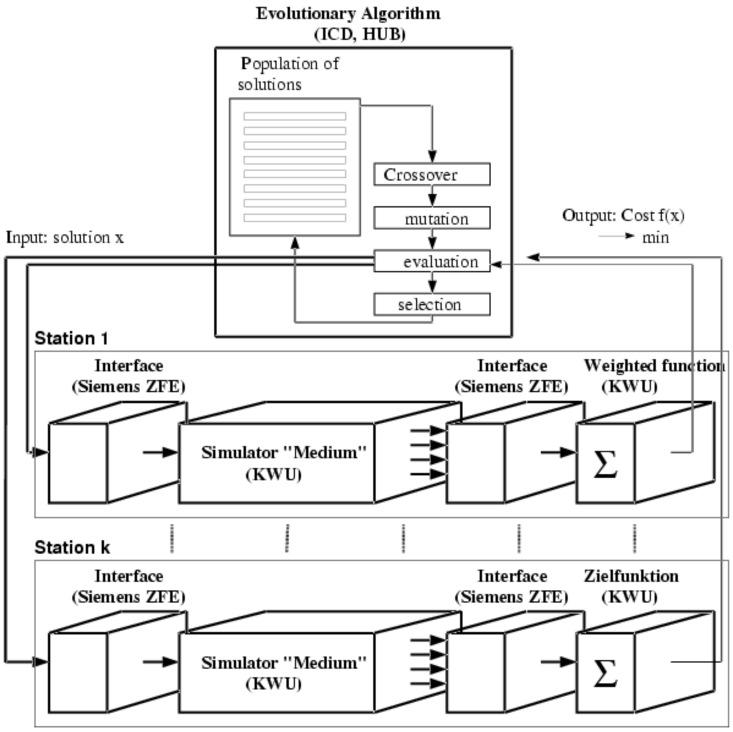


FIG. 8.4 – *Distribution des calculs de performance, (c) Th. Bäck, ICD, Dortmund.*

pas expérimentalement que ce phénomène soit très pénalisant.

Mais plus limitant pour les deux modèles précédents est l'engorgement de communications qui se produit au noeud maître dés que le nombre de processeurs augmente, et si le temps ratio évaluation-évolution n'est pas suffisant : les esclaves doivent alors attendre leur tour pour recevoir le calcul suivant.

Ni Dieu ni maître

Signalons pour mémoire une version totalement tolérante aux pannes – en ce sens que tant qu'un processeur (et le serveur de fichiers) fonctionne, l'algorithme peut continuer – est le modèle dans lequel la population, toujours centralisée, n'est plus maintenue physiquement sur un processeur, mais se trouve stockée dans un fichier auquel tous les processeurs ont accès. Chaque processeur effectue alors l'ensemble des opérations sélection - opérateurs - évaluation - remplacement en lisant et écrivant dans le fichier de la population.

L'inconvénient de ce schéma est bien sûr le temps important nécessaire aux ouvertures/lecture/fermeture du fichier, plus le fait qu'un seul processeur peut écrire en même temps. A n'utiliser qu'en environnement hostile, et si le temps d'un calcul de performance est important.

8.2.3 Modèle en îlots

Le modèle suivant de parallélisation des algorithmes évolutionnaires consiste à diviser la population en petites sous-population, chaque sous-population évoluant sur un processeur, suivant un schéma traditionnel auquel vient s'ajouter une étape de **migration** : chaque sous-population envoie ses meilleurs individus soit vers les populations voisines soit dans un "pool" commun – ce choix dépend du coût relatifs des communications entre processeurs. Chaque sous-population reçoit ensuite des individus soit envoyés par ses voisins soit péchés dans le "pool" central.

Une des possibilités offerte par ce modèle en îlots est que chaque sous-population évolue en utilisant des paramètres différents (voir Figure 8.5) : le réglage des paramètres est un des talons d'Achille des algorithmes évolutionnaires, et c'est là une manière élégante de ne pas choisir ... D'aute part, les "bons" paramètres peuvent être différents à différentes étapes de l'évolution, et on peut espérer qu'une des sous-population au moins aura des paramètres bien réglés à tout moment. En envoyant ses meilleurs individus aux autres sous-populations, elle tirera l'ensemble des individus dans le bon sens.

Mais même lorsque les sous-populations utilisent exactement les mêmes paramètres, la dispersion topologique, fonction du mode de migration choisi (i.e. entre voisins uniquement, ou avec un échange généralisé), a pour effet de préserver la diversité génétique plus longtemps. En effet, il est important de noter qu'il ne s'agit pas ici de la simple parallélisation de l'algorithme séquentiel, mais d'un algorithme à la dynamique totalement différente. Ceci rend en particulier difficiles les comparaisons avec les versions séquentielles, et biaise les tentatives d'estimation du gain.

8.2.4 Population distribuée

Un dernier modèle de parallélisation consiste à distribuer une unique population sur l'ensemble des processeurs (en général sur des machines massivement parallèles). Sur chaque processeur "vivent" alors quelques individus (souvent un seul), et les opérations de sélection/remplacement et de croisement se font entre individus "voisins" pour la topologie du réseau de processeurs.

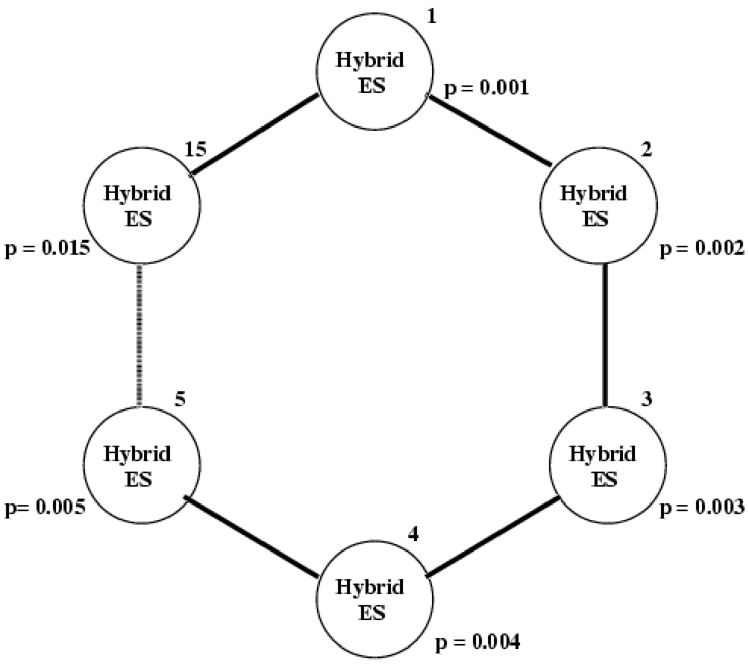


FIG. 8.5 – *Exemple de modéle en îlots – le paramètre de mutation est différent à chaque noeud, (c) Th. Bäck, ICD, Dortmund.*

Ce modèle génère lui aussi une nouvelle dynamique de l'algorithme, différente des précédentes, avec, comme dans le modèle en îlots, l'apparition de "niches écologiques" distribuées par "zones géographiques de processeurs", et permettant donc également la découverte d'optima multiples.

A noter qu'il existe en général dans ce modèle un processeur *superviseur* (ou *root*), qui reçoit des informations de l'ensemble des processeurs et permet de suivre précisément l'évolution de l'ensemble en affichant diverses statistiques. C'est en particulier lui qui décidera de l'arrêt de l'algorithme.

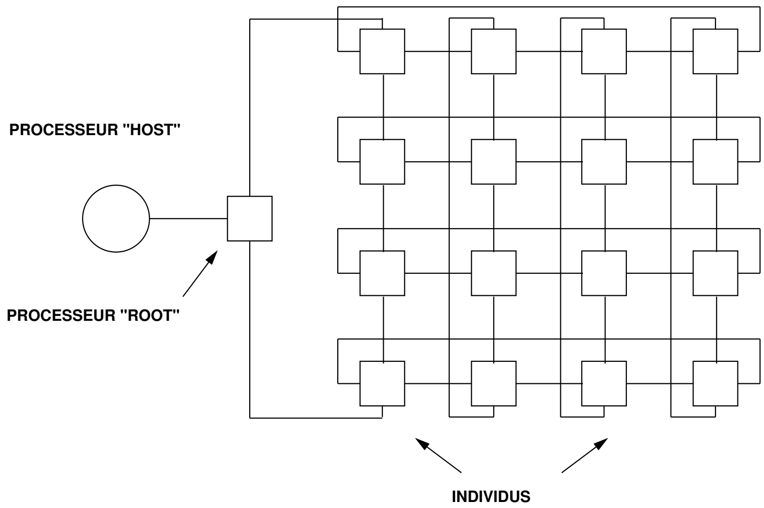


FIG. 8.6 – *Population totalement distribuée, répartie sur une grappe de stations arrangées en topologie torique. Sur chaque noeud se trouvent quelques individus, qui se reproduisent localement.*

8.2.5 Quel modèle choisir?

Il reste certainement des modes de parallélisation des algorithmes évolutionnaires à inventer. Il faut cependant retenir pour le moment que les deux derniers modèles présentés ci-dessus se sont montrés souvent plus performants que l'algorithme séquentiel, au-delà du gain en vitesse du à l'accroissement de ressources (on a constaté des convergences supra-linéaires du modèle totalement distribué par exemple). Au point que ces variantes d'algorithmes sont maintenant quelquefois implantées ... sur des machines séquentielles!

Le choix d'un modèle particulier dépend et du problème et de la configuration matérielle, ... et bien sûr des outils et/ou du temps dont on dispose pour l'implantation.

Négligeant le cas où l'on dispose déjà d'une évaluation de performance parallèle, le modèle le plus simple à implanter est évidemment la distribution du calcul de performance. De plus, il n'y a pas de paramètre supplémentaire à ajuster – une expérience du schéma steady-state séquentiel est suffisante. Le gain à espérer, en cas de réseau de processeurs homogènes, et de calculs équivalents en temps, est strictement linéaire en fonction du nombre de processeurs. Par contre le noeud maître peut rapidement être saturé si le nombre de processeurs est grand en fonction au temps d'un calcul de performance.

Lorsque le nombre de processeurs disponibles augmente, il faut passer au modèle en îlots. L'investissement en développement n'est pas aussi important qu'il y parait à première vue : il faut ajouter à la boucle de génération classique les étapes de migration. Par contre, l'algorithme étant totalement modifié, il faut prévoir un temps de prise en main non négligeable – sans compter les choix de paramètres (stratégie et fréquence de migration). D'un autre côté, on peut espérer des gains plus importants, soit de par l'utilisation de différents jeux de paramètres d'évolution aux différents noeuds, soit de par la dynamique différente – préservation d'une plus grande diversité génétique plus longtemps.

8.3 Résolution de problèmes inverses

Un problème inverse est défini par rapport à un problème direct. Le problème direct consiste à simuler un phénomène physique (ou mécanique, ou chimique), à partir de conditions expérimentales données, d'une loi modélisant le phénomène et d'un schéma numérique :

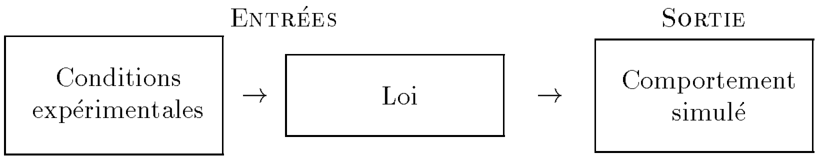


Figure 1 : *Un problème direct*

Le problème inverse consiste, partant de conditions expérimentales et du comportement du phénomène physique observé dans ces conditions, à déterminer une loi modèlisant le phénomène physique de façon satisfaisante. La **performance d'une loi** est donnée par la différence entre le comportement simulé, obtenu par résolution du problème direct pour cette loi avec les conditions expérimentales données, et le comportement réellement observé avec les mêmes conditions expérimentales :

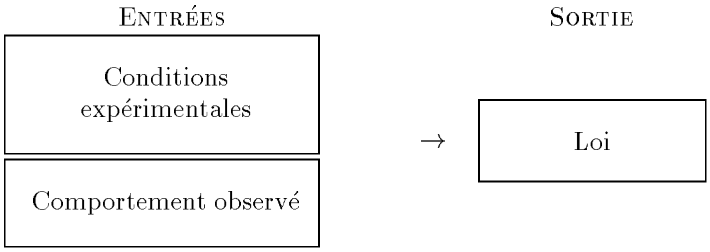


Figure 2 : *Un problème inverse du problème de la Figure 1.*

Fréquemment, les problèmes inverses ne peuvent être traités par les approches classiques faute de disposer d'une fonction performance assez régulière et dont les dérivées soient connues. Au contraire, l'optimisation évolutionnaire requiert uniquement que la fonction performance soit calculable : elle est applicable à tout problème inverse correspondant à un problème direct numériquement résolu.

Reste à choisir l'espace dans lequel rechercher la loi inconnue, ce qui pose le dilemme suivant. Lorsque l'espace exploré est trop restreint, la qualité de la solution peut être compromise ; lorsque cet espace est trop vaste, la convergence de l'optimisation peut être trop lente.

La fin de ce chapitre décrit un exemple en acoustique, l'identification de sous-sol en prospection pétrolière.

8.3.1 Exemple en acoustique

On se propose de retrouver une anomalie dans le sous-sol en fonction d'enregistrements de sismogrammes réalisés en surface d'après des explosions artificielles. La propagation des explosions est simulée par la résolution numérique de l'équation des ondes (avec conditions aux limites absorbantes, voir [Bam98]). Plusieurs représentations sont possibles pour la cible de l'identification :

1. Si l'on suppose que l'on recherche une anomalie dans un sous-sol par ailleurs homogène, que les caractéristiques physiques du sous-sol et de l'anomalie sont connues, et que la forme de l'anomalie est une ellipse, on est ramené à l'identification ...de 2 paramètres, les deux axes de l'ellipse d_1 et d_2.
2. Il est possible également de chercher la valeur de la vitesse de propagation dans cette ellipse. Le problème reste un problème d'optimisation *paramétrique*, i.e. l'espace de recherche est un sous -ensemble de $\mathbb{R}^n$ (ici, $n = 3$).
3. On peut à loisir rajouter des inconnues, par exemple la position du centre de l'ellipse (espace de recherche $= \mathbb{R}^5$).
4. Mais il est plus difficile de rechercher une forme inconnue - voire une distribution totalement arbitraire des vitesses de propagation dans le sous-sol. La première question à résoudre pour ce problème, tout de même plus réaliste, est alors celle du choix de l'espace de recherche.

A noter par contre que quelle que soit la *représentation* choisie, le calcul de la fonction performance est exactement le même, à savoir la résolution d'autant de problèmes directs qu'il y a de sources d'explosion. Par contre, le choix d'une représentation générale implique que la recherche peut être plus longue ...

Dans tous les cas, il suffit (!) de disposer d'opérateurs génétiques (croisement, mutation) en rapport avec le problème et la représentation.

8.3.2 Représentation paramétrique

Le problème est de trouver une représentation pour un champ scalaire (la vitesse de propagation des ondes dans le milieu considéré) dans un domaine 2d. La première approche est "naturelle" pour tout numéricien: On discrétise le domaine souterrain en *éléments*, étape de toute façon nécessaire à la résolution numérique du problème direct, et on considère une variable par élément.

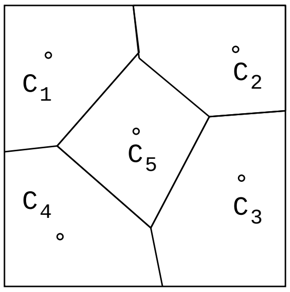
Les opérateurs usuels pour l'évolution de vecteurs de variables réelles sont le croisement arithmétique et la mutation gaussienne adaptative (voir section 8.1.3).

L'inconvénient de cette méthode est le nombre élevé de variables dés que l'on souhaite une précision importante, ainsi que l'absence totale de toute corrélation entre les diverses variables : il est clair que des variations très rapides du sous-sol entre plusieurs éléments voisins ne sont pas possibles physiquement. Par contre, des individus présentant de telles anomalies ont la même probabilité d'apparition dans l'espace de recherche que des configurations de vitesse "raisonnables".

8.3.3 Diagrammes de Voronoï

Avec un peu d'imagination, on peut trouver d'autres représentations à la fois plus réalistes en terme de description du sous-sol, mais surtout dont la complexité (le nombre de "variables") ne dépend pas d'une quelconque discrétisation.

Une approche possible repose sur la notion de *Diagrammes de Voronoî* : l'espace de recherche est l'ensemble des listes de taille variable de *sites de Voronoî* auxquels est associé une vitesse de propagation. Le pavage du sous-sol par les *cellules de Voronoî* correspondantes réalise alors une disposition des vitesses de propagation "constantes par gros morceaux" dans le sous-sol – ce qui est déjà plus réaliste (voir



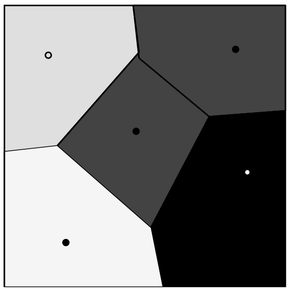


FIG. 8.7 – *Exemple de représentation de Voronoï. Le génotype est la liste des sites de Voronoïmunis chacun d'un scalaire (la vitesse en ce site). Le phénotype est obtenu par construction du diagramme de Voronoïcorrespondant : un point du domaine a la vitesse du site de Voronoïle plus proche de lui.*

Figure 8.7. De plus, la complexité du maillage n'intervient pas sur la taille de l'espace de recherche. Les détails sur cette représentation et les opérateurs associés (qui se devinent aisément), ainsi que des exemples de résultats obtenus sur ce problème inverse (avec un algorithme séquentiel!) peuvent être trouvés dans l'article [SAB98].

Annexe A

Remarques sur la validation de programmes

Sans vouloir (ni pouvoir) être exhaustif, voici quelques indications sur les aspects auxquels il faut faire attention quand on fait du calcul scientifique parallèle. Le premier aspect est celui de s'assurer que la coordination des processus est correcte. Pour montrer la difficulté de la chose, en particulier en mémoire partagée, on propose d'étudier quelques algorithmes classiques (voir aussi [Ray84]) d'exclusion mutuelle. A cette occasion, on utilise (sans trop formaliser, on pourra consulter [Win93] pour plus de détails) quelques notions de preuves à la Hoare. Le deuxième aspect est celui de la validité du calcul numérique proprement dit. On n'a pas insisté sur les aspects les plus classiques de stabilité des schémas etc. (voir par exemple [Bam98]). Ici on a fait le choix d'expliquer comment sont représenter les nombres flottants (qui ne sont pas les réèls mathématiques) en mémoire afin d'attirer l'attention sur les dangers de la programmation numérique.

A.1 Validation de la coordination

A.1.1 En mémoire partagée

Algorithmes d'exclusion mutuelle

Essayons de programmer en pseudo-code une première approximation de programme permettant d'assurer l'exclusion mutuelle. Intéressons-nous d'abord à un premier cas dans lequel on a seulement deux processeurs P_0 et P_1. L'idée est d'assurer que l'on ne puisse jamais être en même temps dans certaine partie du code (appelée section critique) pour P_0 et P_1.

Le premier code auquel on puisse penser pour P_i est ($i = 0, 1$, $j = (i+1) \bmod 2$):

```
VAR turn: 0..1;

LOOP
WAIT UNTIL (turn=i);
   {Section Critique}
turn:=j;
   {Section Non-Critique}
END LOOP
```

Mais dans ce cas les processus ne peuvent entrer en section critique qu'à tour de rôle, ce qui est très limitatif (en particulier si un des processeurs ou un des processus s'arrête sur une erreur pendant sa section critique...).

Pour résoudre cette difficulté il faut s'arranger pour connaître l'état de chaque processus afin de prendre une décision. Un essai de solution serait:

```
VAR flag: ARRAY[0..1] OF BOOL;

WAIT UNTIL (flag(j)=FALSE);
flag(i):=TRUE;
   {Section Critique}
flag(i):=FALSE;
```

Mais hélas maintenant cela n'assure plus l'exclusion mutuelle!

Pour s'en convaincre il suffit de considérer l'exécution suivante:

P_i	P_j	flag(i)	flag(j)
`flag(j)=FALSE`?: OUI	`flag(i)=FALSE`?: OUI	FALSE	FALSE
`flag(i):=TRUE`	`flag(j):=TRUE`	TRUE	TRUE
`Section Critique`	`Section Critique`	-	-

On propose alors le code suivant, toujours plus compliqué...

```
flag(i):=TRUE;
WHILE (flag(j)=TRUE) DO flag(i):=FALSE;
                        WAIT UNTIL NOT(flag(j));
                        flag(i):=TRUE;
END WHILE
   {Section Critique}
flag(i):=FALSE;
```

Il réalise bien l'exclusion mutuelle mais peut causer un interblocage si les deux processus s'exécutent de façon purement synchrone. A ce moment là, après la première affectation, les deux tests de la boucle `WHILE` sont vrais, d'où les deux `flag` sont mis à `FALSE` *simultanément* puis à `TRUE` et ainsi de suite. Les programmes de P_0 et de P_1 bouclent avant la portion `Section Critique`.

Une solution est d'instaurer une règle de *politesse* entre les processeurs. C'est l'algorithme classique décrit dans la section suivante.

L'algorithme de Dekker (1965)

```
VAR flag: ARRAY[0..1] OF BOOL;
    turn: 0..1;

flag(i):=TRUE;
WHILE (flag(j)=TRUE) DO IF turn=j
                        THEN flag(i):=FALSE;
                             WAIT UNTIL NOT(turn=j);
                             flag(i):=TRUE;
                        END IF
END WHILE
   {Section Critique}
turn:=j;
flag(i):=FALSE;
```

Il réalise l'exclusion mutuelle sans blocage mutuel.

L'"algorithme" de Hyman (1966)

Considérons le code plus simple suivant, proposé par Hyman en 1966:

```
VAR flag: ARRAY[0..1] OF BOOL;
    turn: 0..1;

flag(i):=TRUE;
WHILE NOT(turn=i) DO
                     WAIT UNTIL NOT(flag(j));
                     turn:=i;
END WHILE
   {Section Critique}
flag(i):=FALSE;
```

Il est malheureusement faux car, si `turn` est à 0 et P_1 positionne `flag(1)` à `TRUE` puis trouve `flag(0)` à `FALSE`, alors P_0 met `flag(0)` à `TRUE`, trouve `turn` égal à 0 et rentre en section critique. P_1 affecte alors 1 à `turn` et rentre également en section critique!

En conclusion, il n'est pas si facile de programmer avec des processus communiquant par mémoire partagée (cf. JAVA par exemple).

L'algorithme de Peterson (1981)

Une vraie amélioration de l'algorithme de Dekker est l'algorithme de Peterson proposé en 1981.

Contentons nous d'abord du cas simple où l'on dispose uniquement de deux processeurs P_0 et P_1. Le code de P_i est ($i = 0, 1$, $j = (i+1) \bmod 2$):

```
VAR flag: ARRAY[0..1] OF BOOL;
    turn: 0..1;

flag(i):=TRUE;
turn:=i;
while (flag(j)=TRUE et non(turn=j)) do
skip;
od;
   {Critical Section}
flag(i):=FALSE;
```

On a bien alors exclusion mutuelle sans interblocage et avec équité (pas de famine).

Donnons ici quelques idées de preuve de ces assertions.

Introduisons deux variables auxiliaires $after(1)$ et $after(2)$ (qui servent à indiquer si le contrôle est après `turn=i` dans P_i en rajoutant les instructions $flag(1)$=`FALSE` et $flag(2)$=`FALSE` avant la mise en parallèle des deux processus et $after(1)$=`TRUE`, $after(2)$=`TRUE` après respectivement `turn=1` et `turn=2`.

En utilisant l'abbréviation `I= [turn=1 ou turn=2]` on commente le code avec les assertions logiques suivantes:

```
{ non flag(1) }
flag(1)=TRUE; after(1)=FALSE;
{ flag(1) et non after(1) }
turn=1; after(1)=TRUE;
{ inv: flag(1) et after(1) et I }
while (flag(2) et non(turn=2)) do
```

```
   { flag(1) et after(1) et I }
   skip;
od;
{ flag(1) et after(1) et (flag(2) et (non(after(2)) ou turn=2)) }
[ Section Critique CS1]
{ flag(1) }
flag(1)=FALSE;
```

De même pour le deuxième processus,

```
{ non flag(2) }
flag(2)=TRUE; after(2)=FALSE;
{ flag(2) et non after(2) }
turn=2; after(2)=TRUE;
{ inv: flag(2) et after(2) et I }
while (flag(1) et non(turn=1)) do
   { flag(2) et after(2) et I }
   skip;
od;
{ flag(2) et after(2) et (flag(1) et (non(after(1)) ou turn=1)) }
[ Section Critique CS2]
{ flag(2) }
flag(2)=FALSE;
```

Il est relativement aisé de se convaincre ces assertions forment bien un shéma de preuve correct de chacun des processus pris séquentiellement. Seule subsiste la preuve de non-interférence. En fait, seul,

`pre(CS1)=flag(1) et after(1) et (flag(2) et (non after(2) ou (turn=2)))`

contient des références à des objets manipulés par l'autre processus, et ce, seulement en les lignes `flag(2)=TRUE; after(2)=FALSE;` et `turn=2; after(2)=TRUE;` . Or,

```
{ pre(CS1) } flag(2)=TRUE; after(2)=FALSE; { pre(CS1) }
{ pre(CS1) } turn=2; after(2)=TRUE; { pre(CS1) }
```

et de même par symétrie. De plus, on peut voir que

`pre(CS1) et pre(CS2) implique (turn=1 et turn=2)`

Ce qui implique que l'on a toujours

`non(pre(CS1) et pre(CS2))`

prouvant ainsi l'exclusion mutuelle.

Généralisation de l'algorithme de Peterson à n processeurs

Le code suivant, codé sur chacun des n processeurs (P_0 à P_{n-1}) réalise bien l'exclusion mutuelle:

```
VAR flag: ARRAY[0..n-1] OF [-1..n-2]
    turn: ARRAY[0..n-2] OF [0..n-1]

FOR (j=0..n-2) DO
                  flag(i):=j;
                  turn(j):=i;
                  WAIT UNTIL ((all k<>i, flag(k)<j) OR (turn(j)<>i));
END FOR
   { Section Critique }
flag(i):=-1;
```

En fait c'est la solution pour 2 processeurs itérée $n-1$ fois. `flag(i)=` -1 signifie que P_i ne s'engage pas en section critique. `turn` permet de gèrer les conflits dans un couple de processus. Cet algorithme réalise bien l'exclusion mutuelle sans interblocage et avec équité.

Dans les améliorations possibles, il y a en particulier l'algorithme de Burns (1981). C'est une solution entièment symétrique qui minimise le nombre de variables utilisées ainsi que les valeurs qu'elles peuvent prendre en cours de programme.

A.1.2 Algorithmes distribués

Supposons que chacun des p processeurs P_i $(i = 0, \ldots, p-1)$ est muni d'une variable booléenne locale $\texttt{demandeur}_i$ indiquant s'il désire rentrer ou non en section critique. Tous les P_i essaient d'exécuter leur section critique SC_i si et seulement si $\texttt{demandeur}_i$ est vrai. Si $\texttt{demandeur}_i$ est faux alors P_i exécute HSC_i ("Hors Section Critique"). Seul HSC_i peut modifier la variable $\texttt{demandeur}_i$. Il existe alors plusieurs méthodes distribuées permettant d'assurer la propriété de section critique.

Algorithme de LeLann

On organise de façon logique les p processeurs en anneau. On considère un unique *jeton* tournant sur cet anneau. Le processeur qui possède le jeton est seul autorisé à exécuter sa section critique.

Algorithme de Ricart et Agrawala

Lorsqu'un processeur désire entrer en section critique, il demande la permission à chacun des $p-1$ autres processeurs. Une fois qu'il a reçu toutes les permissions, il est autorisé à exécuter sa section critique.

Par exemple dans le cas où p vaut 2 et on définit P_1 et P_2 de la façon suivante:

- P_1 commence par accorder la permission d'entrer en section critique à P_2 si P_1 reçoit une demande de la part de P_2. P_1 demande alors la permission à P_2 d'entrer en section critique. Si elle lui est accordée, alors P_1 rentre en section critique.
- P_2 commence par accorder la permission d'entrer en section critique à P_1 si P_2 reçoit une demande de la part de P_1. P_2 demande alors la permission à P_1 d'entrer en section critique. Si elle lui est accordée, alors P_2 rentre en section critique.

Mais si les demandes et les accords de permissions sont des messages synchrones on a un problème potentiel d'interblocage. Inversement si les demandes et les accords de permissions sont des messages asynchrones il est possible que les deux processeurs rentrent en section critique au même moment!

Une façon de résoudre le problème est de supposer que l'on a une horloge globale (qui peut être par exemple réalisée à partir d'horloges locales). On estampille les demandes par l'heure de départ du message et par l'identité du demandeur, et on peut alors choisir de demander et d'accorder les permissions par des messages asynchrones.

A.2 Validation du calcul numérique

A.2.1 De l'imprécision en général

Un ordinateur calcule essentiellement faux: les entiers machines ne sont pas les entiers mathématiques (ils sont bornés), et les flottants ne sont qu'une représentation approchée finie des nombres réels. Par exemple tous les nombres transcendants (π, e etc.) sont approximés, de même que les rationnels ayant une expansion binaire infinie: attention,

$$\frac{1}{10} = 0.000110011001100110011001100\cdots$$

et est donc tronqué en machine (voir la section suivante pour plus de détails). Les opérations élémentaires ne vérifient pas les lois algébriques usuelles (associativité en particulier) parce-que des arrondis se glissent entre chaque opération. Enfin, le problème de l'imprécision ne touche pas seulement les quelques dernières décimales comme on pourrait le croire. Par exemple $f(a,b) = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$. En toutes précisions sur SUN: $f(77617, 33096) = 1.172603...$, en fait $f(77617, 33096) = -0.827396....$ Le calcul numérique mal compris peut amener à des catastrophes...

A.2.2 La norme IEEE754

C'est la norme de calcul (1985) sur les nombres flottants, à laquelle vous aurez peut-être accès en C (en utilisant des librairies, telles `ieee754.h` sur SUN). Ce qu'il faut retenir c'est que toutes les représentations actuelles sur ordinateur convergent vers un tel standard. Une bonne partie du hardware (sauf quelques Cray) suivent la norme, mais les compilateurs ne sont pas encore à niveau (Microsoft...), en particulier pour la gestion des exceptions arithmétiques ("underflow" etc.).

Représentation en mémoire La norme définit les nombres standards, $r = s * n * 2^{k+1-N}$, avec $s = -1, 1$, $1 - 2^K < k < 2^K$, $-2^N < n < 2^N$ normalisés pour que k soit tel que $r = s * 2^k(1+f)$ avec $f < 1$, en plusieurs versions,

- simple (`REAL*4`, `float`), $K = 7$, $N = 24$,
- double (`REAL*8`, `double`), $K = 10$, $N = 53$ bits,
- double étendu (`REAL*10` etc., `long double`), $K \geq 14$, $N \geq 64$.

Elle définit les nombres dénormalisés (pour gérer les "underflow" de façon graduelle), $r = s * n * 2^{k+1-N} = s * 2^k(0+f)$ avec $k = 2 - 2^K$ et $0 < n < 2^{N-1}$ c.a.d. $0 < f < 1$.

Elle définit également $+\infty$ et $-\infty$ (remarquer que leurs inverses, $+0$ et -0 sont là également).

Enfin, elle définit les `NaN` "Not a Number" signés ou pas (par exemple $0 * \infty$).

Quelques exemples Pour un flottant simple, le maximum normalisé est de 3.402 82347*10^{38}, le minimum normalisé positif est de $1.17549435 * 10^{-38}$, le maximum dénormalisé positif est de $1.17549421 * 10^{-38}$ et le minimum dénormalisé positif est de $1.40129846 * 10^{-45}$.

Autour de 1 l'erreur maximale ("unit in the last place") est de 2^{-23} pour un flottant simple c'est à dire environ $1.19200928955 * 10^{-7}$.

Opérations La norme impose que $+, -, *, /, \sqrt{}$ soient calculés avec une imprécision ne pouvant dépasser la moitié de la distance entre deux flottants du type choisi autour du résultat exact (s'il n'y a pas d'"overflow"). Quand on a le choix entre deux arrondis possibles, on choisit celui qui a une mantisse paire.

On a également accès à des modes d'arrondi,

- arrondis vers zéro (`int`, `aint` etc.)
- arrondis le plus proche, le plus grand en valeur absolu pour les demis (`nint`, `anint` etc.),
- arrondi vers plus l'infini,
- arrondi vers moins l'infini.

A.2.3 Que faut-il faire?

Dans le cadre du calcul scientifique se préoccuper des problèmes de précision de calcul est fondamental. Il faut se poser des questions tant au niveau de schéma purement mathématique de calcul qu'au niveau informatique.

D'un point de vue mathématique, si on a un problème mal conditionné (par exemple le calcul de l'inverse d'une matrice avec un tout petit déterminant), on aura toutes les raisons de craindre le pire numériquement.

D'un point de vue informatique, on peut avoir certaines catastrophes dues à une mauvaise compréhension de l'ordre de grandeur des quantités en jeu.

Dans tous les cas, après une nécessaire réflexion mathématique sur le schéma, il faudra penser à valider numériquement le programme écrit. On pourra par exemple, dans le cas de l'équation des ondes, comparer une solution explicite dans un cas simple à la résolution approchée (par le schéma discret choisi) de cette même solution. L'analyse mathématique du shéma doit donner une indication de l'écart entre les deux méthodes.

A.2.4 Un exemple

On considère le calcul itératif de la puissance $n$ième du nombre d'or $\Phi = \frac{\sqrt{5}-1}{2}$ par deux méthodes differentes,

(1) en utilisant la remarque: $\Phi^n = \Phi^{n-2} - \Phi^{n-1}$,
(2) par calcul direct (multiplication itérée).

Pour la méthode (1):

```
#include <stdio.h>
#include <math.h>

main()
{
  float x,y,z;
  int i;
  x=1;
  y=(sqrt(5)-1)/2;
  for (i=1;i<=20;i++)
    {
      z=x;
      x=y;
      y=z-y;
      printf("phi^%d=%f\n",i,x);
      }
}
```

Résultats:

```
phi^1=0.618034
phi^2=0.381966
phi^3=0.236068
phi^4=0.145898
phi^5=0.090170
phi^6=0.055728
phi^7=0.034442
phi^8=0.021286
phi^9=0.013156
```

```
phi^10=0.008130
phi^11=0.005026
phi^12=0.003103
phi^13=0.001923
phi^14=0.001180
phi^15=0.000743
phi^16=0.000437
phi^17=0.000306
phi^18=0.000131
phi^19=0.000176
phi^20=-0.000045
```

Visiblement fortement erroné...

En fait, à chaque tour de boucle les erreurs sur x, y et z sont données par la transformation affine,

$$\begin{pmatrix} \delta'_x \\ \delta'_y \\ \delta'_z \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \delta_x \\ \delta_y \\ \delta_z \end{pmatrix} + \begin{pmatrix} 0 \\ \alpha \\ 0 \end{pmatrix}$$

Avec au premier tour de boucle $\delta_y = \epsilon = 2^{-23}$. On voit facilement que $\alpha \leq 2^{-24}$ (en fait est de l'ordre de 2^{-23-i}). D'autre part la transformation linéaire sur les erreurs a deux valeurs propres non nulles $\rho_1 = \frac{-1+\sqrt{5}}{2}$ et $\rho_0 = \frac{-1-\sqrt{5}}{2} < -1$!. D'où une erreur à la vingtième itération de l'ordre de $\frac{\epsilon}{\rho_1-\rho_0}\rho_1^{20} + \frac{|\rho_1^{21}|-1}{\rho_1-1}\alpha$ (environ $4*10^{-4} + 2.4*10^{-3}$) supérieur à ρ_0^{20} (environ $6.6*10^{-5}$).

Maintenant, au tour du programme suivant la méthode (2), d'ailleurs pas nécessairement plus lent:

```
#include <stdio.h>
#include <math.h>

main()
{
  float t,p;
  int i;
  t=1;
  p=(sqrt(5)-1)/2;
  for (i=1;i<=20;i++)
    {
      t=t*p;
      printf("phi^%d=%f\n",i,t);
      }
}
```

Résultats:

```
phi^1=0.618034
phi^2=0.381966
phi^3=0.236068
phi^4=0.145898
phi^5=0.090170
phi^6=0.055728
phi^7=0.034442
phi^8=0.021286
```

```
phi^9=0.013156
phi^10=0.008131
phi^11=0.005025
phi^12=0.003106
phi^13=0.001919
phi^14=0.001186
phi^15=0.000733
phi^16=0.000453
phi^17=0.000280
phi^18=0.000173
phi^19=0.000107
phi^20=0.000066
```

Il est facile de voir que l'erreur à la vingtième itération est de l'ordre de $[\rho_0 - 2^{-23}, \rho_0 + 2^{-23}]^{20}$ c'est à dire environ $[6.61067063328 * 10^{-5}, 6.61072163724 * 10^{-5}]$.

Dans le domaine du calcul scientifique, on retrouvera de tels systèmes dynamiques (pour des schémas explicites). Heureusement, la plupart du temps, la condition de stabilité des schémas assurera une non-dégradation des erreurs. Pour les schémas implicites, on aura à résoudre des systèmes linéaires. Là, selon les méthodes utilisées, on peut avoir des choix algorithmiques meilleurs que d'autres en termes de précision de calcul. Par exemple, on choisira plutôt le pivot le plus grand en norme dans la méthode d'élimination de Gauss.

Annexe B

Autres librairies

B.1 PVM3.4

La version 3.4 est la toute dernière version de PVM. Elle ajoute quelques fonctionnalités type MPI, et en particulier:

- Contextes de communication
- Handlers
- Messages persistants
- etc.

B.1.1 Contextes de Communication

C'est un "tag" supplémentaire qui permet en particulier de bien filtrer les messages utilisateurs des messages systèmes (ou entre groupes, ou entre applications différentes). Pour ces nouveaux "tag", il n'y a pas de possibilité de "wildcard" (filtrage à la réception avec TAG=-1).

Ces "tags" assurent un certain nombre de propriétés. Toute création d'un nouveau contexte définit un unique tag pour toute une machine PVM. De plus les tâches filles héritent du contexte du père.

A noter que cette extension de PVM ne rend pas obsolète les anciens programmes PVM. Les principales commandes associées sont:

- `new_context=pvm_newcontext()` crée un nouveau contexte pour le reste du code,
- `old_context=pvm_setcontext(new_context)` change le contexte à `newcontext` et renvoie l'ancien contexte,
- `info=pvm_freecontext(context)` libère le tag `context`,
- `context=pvm_getcontext()` renvoie le contexte courant.

Dans la pratique, une tâche demande un nouveau contexte et le distribue à toutes les autres taches par `newcontext=pvm_newcontext()`. Puis les autres tâches se mettent au nouveau contexte pendant les appels aux fonctionnalités privées de la première tâche par:

```
oldcontext=pvm_setcontext(newcontext);
/* communications sures */
newcontext=pvm_setcontext(oldcontext);
pvm_freecontext(newcontext);
/* on revient a l'ancien contexte */
```

B.1.2 Handlers

Les "handlers" permettent d'exécuter une fonction C à la réception d'un message spécifique. Ils agissent comme des interruptions (qui n'interrompent le cours du programme que uniquement pendant les appels à la librairie PVM). La fonction C (il n'y a aucune restriction d'usage) est appelée quand un message correspond à un pattern (*context*, *msgtag*, *source*) donné.

Les principales commandes correspondantes sont:

- `mhid=pvm_addmhf(src,tag,context,func)` déclare le handler `func` (n'importe quelle fonction C) à activer à la réception de tout message compatible avec le pattern `src,tag,context`.
- `pvm_delmhf(mhid)` détruit le handler `mhid`.

B.1.3 Messages persistants

Les messages persistants sont des messages "uni-directionnels", similaire au "tuple-space" de Linda. Les tâches PVM peuvent enregister et récupérer des messages par leurs noms: cela organise de fait une base de données distribuée pour les processus. Il est également possible de ranger plusieurs messages sous un même nom.

Les principales commandes sont:

- `index=pvm_putinfo(name,msgbuf,flag)` qui range dans la base de données un message sous le nom `name`,
- `index=pvm_recvinfo(name,index,flag)` qui extrait de la base de données un message de nom `name`,
- `index=pvm_delinfo(name,index,flag)` qui supprime de la base de données un message de nom `name`.

B.2 MPI

MPI ("*M*essage *P*assing *I*nterface") est le prochain standard de fait (voir `http://www.mcs.anl.gov/mpi/`). MPI fait l'objet fin 1992 d'un groupe de travail de 40 experts industriels et académiques. Le premier standard MPI-1 est réalisé fin 1993. Le principe est que chaque vendeur de supercalculateurs implémente sa propre API propriétaire en respectant le standard. Cela fait que MPI est particulièrement bien optimisé sur un grand nombre d'architectures de super-calculateurs. Mais le système n'est pas basé comme PVM sur la notion de machine virtuelle, et les implémentations sont moins bien étudiées pour les réseaux de stations (éventuellement hétérogènes). L'abstraction utilisée par MPI est celle de topologie de passage de messages. La deuxième version du standard a commencé début 1995, et le premier brouillon a été publié fin 1996.

B.2.1 MPI-1

Le premier standard décrit 128 fonctions, mais pas d'équivalent de `pvm_spawn`. Les processus sont donc définis de façon statique uniquement. MPI-1 comprend une notion de `communicator` englobant les notions de contextes (de PVM3.4) et les groupes de processus (de PVM3.3).

La grande force du standard est le fait qu'il inclut énormément de fonctions de communication point à point et entre groupes de processus. Ces fonctions reposent sur des fonctions de définition de topologies de communication ainsi que de définitions de `datatypes` qui décrivent des messages pris à des endroits non contigus de la mémoire.

B.2.2 MPI-2

Le deuxième standard décrit 120 fonctions de plus et en particulier des fonctions de communication collectives non-bloquantes et un `MPI_SPAWN` (équivalent de `pvm_spawn`).

MPI-2 intègre également des communications unidirectionnelles (les messages persistants de PVM3.4) et l'équivalent des `handlers` de PVM3.4. MPI-2 devrait avoir des fonctions pour gérer la tolérance aux pannes.

Il existe quelques versions gratuites actuellement implémentant en partie les standards MPI dont,

- LAM (version 6.3 et 6.4a) qui implémente MPI-1 plus quelques fonctionnalités MPI-2 (`MPI_SPAWN` etc.), voir `http://www.mpi.nd.edu/lam/`
- MPICH (version) qui implémente strictement MPI-1, voir `http://www.mcs.anl.gov/mpi/mpich/` et `ftp://info.mcs.anl.gov/pub/mpi`

Sinon, il existe des versions contructeurs (non gratuites...) sur CRAY, SGI, IBM etc.

B.2.3 Exemple

On donne ci-après un petit exemple de code MPI (fourni avec la distribution standard).

```
/* Transmet un message dans un systeme a 2 processus. */
#include <mpi.h>
#define BUFSIZE 64
int buf[BUFSIZE];
int main(argc, argv)
     int argc;
     char *argv[];
{
  int size, rank;
  MPI_Status status;
  /* Initialisation de MPI. */
  MPI_Init(&argc, &argv);
  /* On verifie le nombre de processus presents */
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (2 != size) {
    MPI_Finalize();  /* comme pvm_exit */
    return(1);
  }

  /* Determine son rang dans le groupe "World" */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* L'envoyeur aura rang 0, le receveur, rang 1 */
  /* Si rang 0 envoyer au rang 1 */
  if (0 == rank) {
    MPI_Send(buf, BUFSIZE, MPI_INT, 1, 11, MPI_COMM_WORLD);
  }
  /* Si rang 1 recevoir du rang 0 */
  else {
    MPI_Recv(buf,BUFSIZE,MPI_INT,0,11,MPI_COMM_WORLD,&status);
  }
```

```
  MPI_Finalize();
  return(0);
}
```

Exemple:

```
#include <mpi.h>
#define WORKTAG 1
#define DIETAG 2
#define NUM_WORK_REQS 200
static void master();
static void slave();

/* Ce programme est MIMD, mais est ecrit SPMD pour */
/* etre plus simple a lancer */
int main(argc, argv)
     int argc;
     char *argv[];
{
  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,/* groupe de tout le monde */
&myrank); /* 0 a N-1 */

  if (myrank == 0) {
    master();
  } else {
    slave();
  }

  MPI_Finalize();
  return(0);
}
/* Le maitre envoie des demandes de travaux aux */
/* esclaves puis collecte les resultats */
static void master()
{
  int ntasks, rank, work;
  double result;
  MPI_Status status;

  MPI_Comm_size(MPI_COMM_WORLD,&ntasks); /* #processus */
  /* Initialiser les esclaves */
  work = NUM_WORK_REQS; /* simulation */
  for (rank = 1; rank < ntasks; ++rank) {
    MPI_Send(&work,/* tampon de messages */
      1,/* un seul element */
      MPI_INT,/* de ce type */
      rank,/* au proc de ce rang */
      WORKTAG,/* message "travail" */
      MPI_COMM_WORLD);/* contexte */
    work--;
  }
```

```
  /* Recoit un resultat de n'importe quel esclave et */
  /* renvoie un nouveau travail jusqu'a ce que les */
  /* requetes soient epuisees */
  while (work > 0) {
    MPI_Recv(&result,/* tampon de messages */
     1,/* un seul element */
     MPI_DOUBLE,/* de ce type */
     MPI_ANY_SOURCE,/* de n'importe qui */
     MPI_ANY_TAG,/* n'importe quel message */
     MPI_COMM_WORLD,/* communicator */
     &status); /* info */

    MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
     WORKTAG, MPI_COMM_WORLD);

    work--; /* simulation */
  }
  /* Recoit les resultats des esclaves */
  for (rank = 1; rank < ntasks; ++rank) {
    MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  }
  /* Dit aux esclaves de terminer */
  for (rank = 1; rank < ntasks; ++rank) {
    MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
  }
}
/* Chaque esclave accepte les requetes de travaux et */
/* retourne les resultats jusqu'a requete terminaison */
static void slave()
{
  double result;
  int work;
  MPI_Status status;
  for (;;) {
    MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
     MPI_COMM_WORLD, &status);
    /* Verifie le tag du message */
    if (status.MPI_TAG == DIETAG) {
      return;
    }
    sleep(2);
    result = 6.0; /* simulation */
    MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  }
}
```

B.3 Comparaison PVM, MPI

En général MPI a de meilleures performances que PVM (mais elles restent quand même très comparables même sur un Cray T3D ou un IBM SP2). PVM sacrifie les performances pour être portable sur un réseau hétérogène. Par contre MPI est non portable sur réseau de stations (pas de méthode standard pour démarrer MPI, ni de notion de machine virtuelle). MPI a plus de primitives de communications, mais il y a encore un certain nombre de problèmes non-résolus pour les pannes (en particulier vis-à-vis des `communicators`).

En conclusion, PVM3.4 et MPI-2 se ressemblent maintenant beaucoup, aux performances près. Parmi les projets en cours, on trouve d'ailleurs un projet de "convergence" PVM et MPI qui se nomme PVMMPI.

Annexe C

Utilisation de PVM et de XPVM

C.1 Configuration de PVM

Vous avez accès à une partition disque sans quota sous `/users/pvm/LOGIN`. Attention à ne pas laisser trop d'exécutables chez vous! Si vous choisissez d'utiliser cette partition, il faut remplacer les `$HOME` dans tous les fichiers qui suivent (`xpvm_hosts`, `Makefile.aimk`) par `/users/pvm/LOGIN`.

Vous trouverez dans `~goubaul1/pvm_install.tar` une installation prédéfinie pour commencer vos TDs. Récupérez la:

```
> cp ~goubaul1/pvm_install.tar .
> tar -xvf pvm_install.tar
...
> cd pvm
> ls
bin          include     lib          src           xpvm_hosts
```

Le répertoire `bin` contient des répertoires par architecture possible. En effet, on doit avoir un exécutable différent du même programme pour toutes les architectures d'une machine PVM.

Le répertoire `src` contient les sources des programmes PVM. C'est là que vous allez programmer.

Le fichier `xpvm_hosts` est un fichier de configuration de la machine PVM (par défaut). Faites,

```
> cp xpvm_hosts ../.xpvm_hosts
```

Ainsi le fichier de configuration sera automatiquement reconnu par `xpvm` (mais pas automatiquement par `pvm`).

Ce fichier est comme suit,

```
* ep=$HOME/pvm/bin/$PVM_ARCH
& ferrari
& rolls
& pontiac
& cher
& morbihan
& carmor
```

`ep` indique de considérer ce qui suit comme étant l'endroit en machine où chercher les exécutables. Les noms sur les lignes suivantes sont des noms de stations que l'on veut pouvoir inclure dans la configuration de la machine pvm dès son démarrage (marche sous xpvm).

C.2 Compilation d'un programme C/PVM

Pour compiler un programme PVM, il faut,

- compiler le programme C correspondant pour chaque architecture des machines entrant dans la composition de la machine PVM,
- faire le lien avec les bibliothèques PVM,
- ranger le (ou les) exécutable(s) dans des répertoires accessibles aux démons `pvmd3` selon les architectures.

Tout cela est géré par des `makefile`.

Les `makefile` de pvm sont très proches de ceux utilisés ordinairement en `C`. Seule la commande pour les utiliser change: elle s'appelle ici `aimk`.

Exemple de `Makefile` (fourni),

```
SDIR = $(HOME)/pvm/src
#BDIR = $(HOME)/pvm3/bin
BDIR = $(SDIR)/../bin
XDIR = $(BDIR)/$(PVM_ARCH)

OPTIONS = -O
CFLAGS = $(OPTIONS) -I../include -I$(PVM_ROOT)/include $(ARCHCFLAGS)

LIBS = -lpvm3 $(ARCHLIB)
GLIBS = -lgpvm3

LFLAGS = $(LOPT) -L$(PVM_ROOT)/lib/$(PVM_ARCH)

default: hello hello_other

hello: $(SDIR)/hello.c $(XDIR)
$(CC) $(CFLAGS) -o $@ $(SDIR)/hello.c $(LFLAGS) $(LIBS)
mv $@ $(XDIR)

hello_other: $(SDIR)/hello_other.c $(XDIR)
$(CC) $(CFLAGS) -o $@ $(SDIR)/hello_other.c $(LFLAGS) $(LIBS)
mv $@ $(XDIR)
```

Le `makefile` plus haut se contente essentiellement de compiler différents programmes exemples (voir section C.5) avec la librairie `pvm3` puis de bouger le code compilé dans le répertoire que pvm connaît comme renfermant les exécutables (cf `ep` de la section C.1).

Exemple: sur une DEC (tarse):

```
goubaul1@tarse ~ > cd pvm/src
goubaul1@tarse ~/pvm/src > aimk
making in ALPHA/ for ALPHA
cc -O -I../include -I/usr/local/pvm/include -DHASSTDLIB -o hello
/users/profs/goubault/pvm/src/hello.c  -L/usr/local/pvm/lib/ALPHA
-lpvm3
mv hello /users/profs/goubault/pvm/src/../bin/ALPHA
cc -O -I../include -I/usr/local/pvm/include -DHASSTDLIB -o
hello_other /users/profs/goubault/pvm/src/hello_other.c
```

```
-L/usr/local/pvm/lib/ALPHA -lpvm3
mv hello_other /users/profs/goubault/pvm/src/../bin/ALPHA
```

Puis sur une HP (royce):

```
goubaul1@royce ~ > cd pvm/src
goubaul1@royce ~/pvm/src > aimk
making in HPPA/ for HPPA
        cc -O -I../include -I/usr/local/pvm/include -DSYSVSIGNAL
-DNOWAIT3 -DNOGETDTBLSIZ -DRSHCOMMAND=\"/usr/bin/remsh\"
-DFDSETISINT -o hello /users/profs/goubault/pvm/src/hello.c
-Wl,-aarchive -L/usr/local/pvm/lib/HPPA -lpvm3
        mv hello /users/profs/goubault/pvm/src/../bin/HPPA
        cc -O -I../include -I/usr/local/pvm/include -DSYSVSIGNAL
-DNOWAIT3 -DNOGETDTBLSIZ -DRSHCOMMAND=\"/usr/bin/remsh\"
-DFDSETISINT -o hello_other
/users/profs/goubault/pvm/src/hello_other.c
-Wl,-aarchive -L/usr/local/pvm/lib/HPPA -lpvm3

        mv hello_other /users/profs/goubault/pvm/src/../bin/HPPA
```

C.3 La console PVM

```
goubaul1@tarse ~/pvm/src > pvm ~/.xpvm_hosts
```

Les différentes commandes à la console sont

- `conf` permet de lister les différentes machines composant la machine pvm. Par exemple,

```
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
  tarse.polytechnique.fr    40000    ALPHA    1000
```

- `add` permet d'ajouter une station à la machine pvm. Par exemple,

```
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
  tarse.polytechnique.fr    40000    ALPHA    1000
pvm> add radius
1 successful
                    HOST     DTID
                  radius    80000
pvm> conf
2 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
  tarse.polytechnique.fr    40000    ALPHA    1000
                  radius    80000    ALPHA    1000
pvm> add royce
1 successful
                    HOST     DTID
                   royce    c0000
pvm> conf
3 hosts, 2 data formats
                    HOST     DTID     ARCH   SPEED
  tarse.polytechnique.fr    40000    ALPHA    1000
```

```
radius    80000    ALPHA    1000
 royce    c0000     HPPA    1000
```

- `delete` permet de faire exactement le contraire et d'enlever une machine à la configuration pvm.
- `spawn` permet de lancer une ou plusieurs copies d'un programme pvm. Par exemple `spawn ->hello` permet de lancer une copie du programme `hello` et de rediriger les sorties du programme (où que ce processus soit exécuté) vers la console pvm. `spawn ->2 hello` permet d'en lancer deux copies. Exemple:

```
pvm> spawn -> hello
[1]
1 successful
t80001
pvm> [1:t80001] i'm t80001
[1:t80001] from tc0001: hello, world from royce
[1:t80001] EOF
[1:tc0001] EOF
[1] finished
pvm> spawn -> 10 hello
[3]
10 successful
t40009
t4000a
t4000b
t4000c
t80009
t8000a
t8000b
tc000a
tc000b
tc000c
pvm> [3:t4000d] EOF
[3:t40009] i'm t40009
[3:t40009] from t4000d: hello, world from tarse.polytechnique.fr
[3:t40009] EOF
[3:t4000c] i'm t4000c
[3:t4000c] from t4000f: hello, world from tarse.polytechnique.fr
[3:t4000f] EOF
[3:t4000c] EOF
[3:t8000a] i'm t8000a
[3:t8000a] from tc000e: hello, world from royce
[3:tc000a] i'm tc000a
[3:tc000a] from tc000d: hello, world from royce
[3:t8000a] EOF
[3:tc000d] EOF
[3:t4000e] EOF
[3:tc000a] EOF
[3:t8000c] EOF
[3:tc000e] EOF
[3:t8000d] EOF
[3:tc000c] i'm tc000c
[3:tc000c] from t8000c: hello, world from radius.polytechnique.fr
[3:t80009] i'm t80009
[3:t80009] from t8000d: hello, world from radius.polytechnique.fr
```

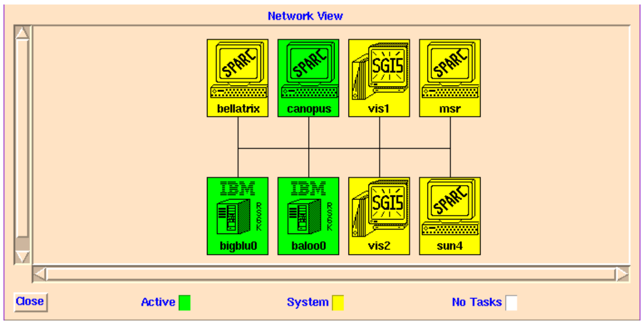


FIG. C.1 – *Machine pvm*

```
[3:tc000c] EOF
[3:t80009] EOF
[3:t4000b] i'm t4000b
[3:t4000b] from tc000f: hello, world from royce
[3:t4000b] EOF
[3:t80009] EOF
[3:t4000b] i'm t4000b
[3:t4000b] from tc000f: hello, world from royce
[3:t4000b] EOF
[3:t8000e] EOF
[3:t4000a] i'm t4000a
[3:t4000a] from t8000e: hello, world from radius.polytechnique.fr
[3:tc000f] EOF
[3:t4000a] EOF
[3:t8000b] i'm t8000b
[3:t8000b] from t4000e: hello, world from tarse.polytechnique.fr
[3:t40010] EOF
[3:tc000b] i'm tc000b
[3:tc000b] from t40010: hello, world from tarse.polytechnique.fr
[3:t8000b] EOF
[3:tc000b] EOF
[3] finished
```

- `quit` permet de quitter la console pvm sans arrêter les démons `pvmd`.
- `halt` quitte la console pvm et arrête les démons `pvmd`.

C.4 L'interface XPVM

XPVM est la version X-Windows de pvm. Faire `xpvm` au clavier pour démarrer la console xpvm, mais surtout pas `xpvm &`. Au démarrage, on voit la figure C.1.

C'est la représentation de la machine PVM avec les noms et les types des différents ordinateurs la composant. Les menus de cette fenêtre permettent de contrôler les tâches (menu `Tasks`), de contrôler les machines (menu `Hosts`), d'arrêter PVM (menus `Reset, Quit, Halt`) et d'avoir de l'aide (menu `Help`). Voir figure C.2.

On peut ajouter et enlever des hôtes (`add` et `delete` de PVM) par le menu `Tasks`. On conseille dans les prochains TDs de commencer par tester les programmes sur un seul hôte (la machine sur laquelle vous vous êtes loggée) puis sur deux ou trois, choisies parmi les voisins immédiat. Cela permettra de ne pas surcharger certaines machines.

Le `spawn` de la console PVM se retrouve dans le menu `Tasks`.

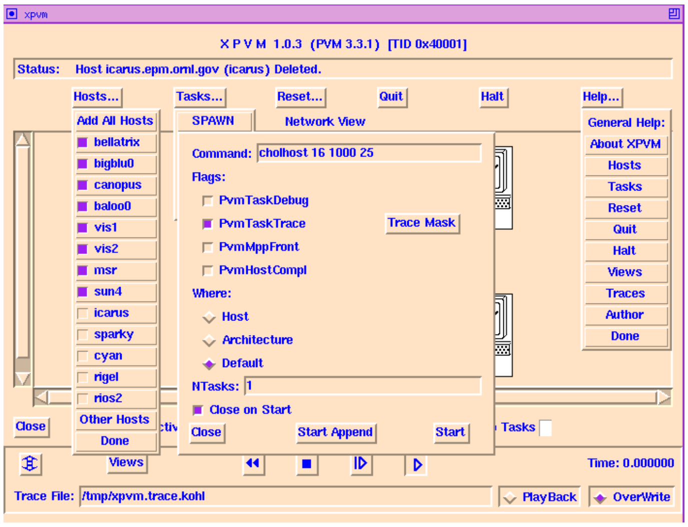


FIG. C.2 – *Menus XPVM*

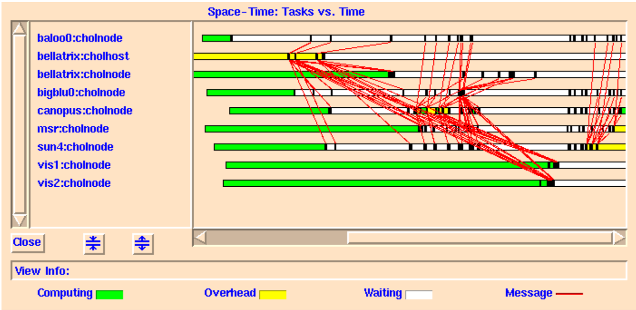


FIG. C.3 – *Activité des tâches*

Quand on exécute une tâche, on voit dans la partie basse de la fenêtre XPVM le déroulement dans le temps des différentes tâches, voir figure C.3.

Chaque couleur représente le type d'activité d'une tâche (“computing”, “overhead”, “waiting”, “messages”). Vous pouvez cliquer sur les flèches représentant les messages ou sur les intervalles de temps pour avoir plus d'informations. Vous pouvez également changer d'echelle de visualisation en cliquant-glissant sur cette fenêtre.

On récupère ce qu'affichent les différentes tâches en demandant dans le menu `View` les “Task Output”, voir figure C.4.

C'est ce que l'on voyait sous la console PVM (quand on avait mis l'option ->.

Dans le même menu on peut également demander un graphe du type d'activité des processus en fonction du temps (“Utilization versus Time”), voir figure C.5.

Ou encore, à voir tous les envois et les réceptions de messages entre les tâches (“Call Trace”), voir figure C.6.

On peut tout avoir en même temps, voir figure C.7.

C.5 Un programme exemple

Ces programmes vous sont fournis dans la distribution `pvm_install.tar`.

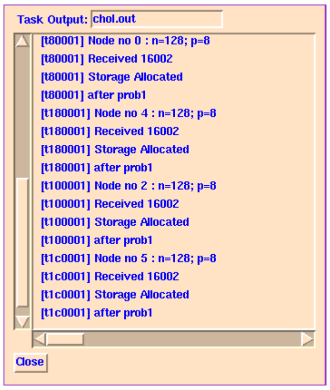


FIG. C.4 – *Task Output*

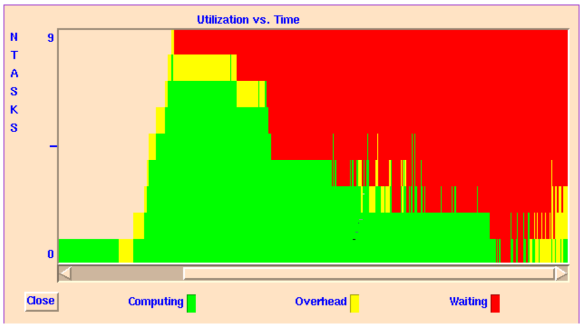


FIG. C.5 – *Utilization versus Time*

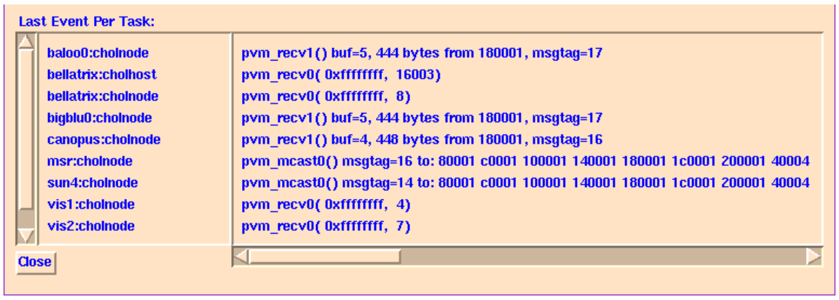


FIG. C.6 – *Call Trace*

FIG. C.7 – *Un écran XPVM au complet*

Le maître (`hello.c`),

```
#include <stdio.h>

/* pour avoir acces a la librairie PVM: */
#include "pvm3.h"

main()
{
int cc, tid;
char buf[100];

/* pour raccrocher le programme au demon pvmd3 */
printf("i'm t%x\n", pvm_mytid());

/* pour lancer un esclave en parallele */
cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);

/* s'il s'est bien lance... */
if (cc == 1)
{
  /* recevoir un message de n'importe qui */
  cc = pvm_recv(-1, -1);
  /* pvm_bufinfo(cc, (int*)0, (int*)0, &tid); */
  /* decompacter ce message et ranger une chaine de */
  /* caracteres dans la variable buf */
  pvm_upkstr(buf);
  printf("from t%x: %s\n", tid, buf);
} else
```

```
  printf("can't start hello_other\n");

/* deconnecter le processus du demon pvmd3 */
 pvm_exit();
 exit(0);
}
```

Le programme esclave (`hello_other.c`),

```
/* pour avoir acces a la librairie PVM: */
#include "pvm3.h"

main()
{
  int ptid;
  char buf[100];

  /* trouver le processus parent */
  ptid = pvm_parent();

  strcpy(buf, "hello, world from ");
  gethostname(buf + strlen(buf), 64);

  /* initialiser un tampon de communication */
  pvm_initsend(PvmDataDefault);
  /* y ranger la chaine de caracteres buf */
  pvm_pkstr(buf);
  /* envoyer le contenu de ce tampon */
  pvm_send(ptid, 1);

  /* deconnecter le processus du demon pvmd3 */
  pvm_exit();
  exit(0);
}
```

Bibliographie

[Bam98] A. Bamberger. *Ondes Acoustiques et Electromagnétiques.* Cours de l'Ecole Polytechnique, 1998.

[BFM97] Th. Back, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation.* Oxford University Press, 1997.

[BNKF97] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming — An Introduction, On the Automatic Evolution of Computer Programs and Its Applications.* Morgan Kaufmann, 1997.

[CS98] D. E. Culler and J. P. Singh. *Parallel Computer Architecture.* Morgan-Kaufmann, 1998.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[LER92] T. G. Lewis and R. El-Rewini. *Introduction to Parallel Computing.* Prentice-Hall, 1992.

[LL] B. Larrouturou and P. L. Lions. *Méthodes mathématiques pour les sciences de l'ingénieur: Optimisation et analyse numérique.* Cours de l'Ecole Polytechnique,??

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[Mic96] Z. Michalewicz. *Genetic Algorithms+Data Structures = Evolution Programs.* Springer Verlag, 1992–1996.

[Ray84] M. Raynal. *Algorithmique du parallélisme: le problème de l'exclusion mutuelle.* Dunod, 1984.

[SAB98] M. Schoenauer, Ehinger A., and B. Braunschweig. Non-parametric identification of geological models. In *Proceedings of the Fifth IEEE International Conference on Evolutionary Computation*, pages 136–141. IEEE Press, 1998.

[Win93] G. Winskel. *The formal semantics of programming languages.* The MIT Press, 1993.