Parallélisation d'algorithmes variationnels d'assimilation de données en météorologie

Yannick Tremolet

Thèse

présentée par

Yannick Trémolet

pour obtenir le titre de Docteur

de l'Université Joseph Fourier - Grenoble 1

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : Mathématiques appliquées

Parallélisation d'algorithmes variationnels d'assimilation de données en météorologie

---

Date de soutenance : 27 Novembre 1995

Composition du jury

Président :	Patrick Witomski
Rapporteurs :	Bernard Philippe
	Michael Navon
Examinateurs :	Brigitte Plateau
	Denis Trystram
	François-Xavier Le Dimet

Thèse préparée au sein du
Laboratoire de Modélisation et Calcul
(Institut de Mathématiques Appliquées de Grenoble)

Remerciements

Cette thèse est le résultat de trois années de travail, je remercie ici toutes les personnes avec qui j'ai partagé cette période.

Tout d'abord, je tiens à remercier les membres du jury et en premier lieu Patrick Witomski qui a accepté de le présider et qui m'a accueilli au sein du Laboratoire de Modélisation et Calcul.

Merci à Michael Navon qui a accepté d'être l'un des rapporteurs de ce travail pour ces judicieux conseils bibliographiques et son soutien dans ma recherche de post-doc.

Je remercie Bernard Philippe qui a également accepté de faire un rapport sur mon travail.

Merci aussi à Brigitte Plateau, la grande chef Apache, qui a bien voulu participer à l'évaluation de mon travail de recherche.

Merci également à Denis Trystram qui, quelque part, m'a permis de venir à Grenoble en DEA et qui m'a accueilli dans l'équipe de calcul parallèle.

Enfin, le dernier mais non des moindres, merci à mon « big chef » François-Xavier Le Dimet pour tout ses conseils éclairés et pour tout ce qu'il m'a fait découvrir : la météorologie, les modèles adjoints, les Houches et les américains.

Merci également à Jacques Blum qui dirige le projet IDOPT dont ce travail fait partie.

Je remercie aussi tous ceux qui tout au long de ces trois ans ont contribué à rendre agréable la vie au labo. D'abord les permanents : le grand Jacques pour ses réflexions politiques et œnologiques, Jacques pour les histoires de l'Histoire de l'informatique, Denis pour ses pépins de pomme et son esprit « zen », Jean-Louis pour son enthousiasme scientifique et humain à toute épreuve, Gilles un « parrain » du sud avec l'humour en plus, Brigitte la « professeuse » de l'équipe, Jean-Marc pour ses bouquins de math et les discussions gastronomiques, Joëlle qui subit nos râleries quand cosmos plante, nos secrétaires Angèle puis Khadija, et l'équipe de calcul formel Claire, Françoise, Abdel, Jean-Baptiste et les autres...

Merci aussi aux thésards, passés et présents, qui ont supporté les personnes citées ci-dessus avec moi : Nathalie pour ses mousses au chocolat et son dévouement à la cause de l'orthographe et de la Kfet, Fred ex-« monsieur galère » maintenant papa, Titou pour le ski, l'escalade, les gâteaux, les pizzas aux champis et Libé, Xtof pour

les cigales et le pastis, Cécile que j'entends rouspéter d'ici, Lolo l'auvergnat sympa, Phil rugbyman entre deux blessures, Pascal hips, Alain qui torture le chocolat avec des pâtes et du poisson, Yves control-alt-Z-k-#, Alex qui a fini par craquer, Christophe qui fait vivre le labo la nuit, Martha, René, Luc, Pierre-Éric, Éric, Matthieu, les brésiliens qui réchauffent l'ambiance générale, et tous les autres... Bon courage à ceux qui n'ont pas encore fini.

Merci à ceux qui m'ont supporté dans leur bureau : Lolo, Xtof et Pierre-Éric ou chez eux en cette fin d'année : Martine et Sébastien, Véro, Gilles et particulièrement Nathalie et Thierry pour leur accueil chaleureux malgré l'eau froide.

Désolé pour tous les skieurs, randonneurs et marins qui veulent des prévisions météo fiables, je n'ai que celles d'hier...

Table des matières

Introduction

Durant les dernières décennies, nous avons assisté à une demande croissante de prévisions de l'évolution de l'état de l'atmosphère et, plus récemment, de l'océan. La demande devient également de plus en plus variée : évolution sur une région précise et à court terme mais avec une grande précision (par exemple autour des aéroports, dans les zones agricoles à certaines périodes cruciales ou pour les activités de loisir en montagne ou en mer) ou évolution à long terme en climatologie pour étudier l'impact de l'activité humaine par exemple. On peut prévoir avec une relative certitude que ces besoins continueront à augmenter avec une demande de précision accrue ou avec la modélisation de phénomènes non pris en compte jusque là : couplage océan-atmosphère, étude de la pollution, étude de la biosphère, etc...

Pour résoudre ce type de problèmes, nous disposons d'observations de l'atmosphère et de l'océan qui sont de plus en plus nombreuses et variées ainsi que de modèles théoriques d'évolution des fluides géophysiques. Ces modèles, au fur et à mesure de l'avancement de nos connaissances, deviennent de plus en plus fiables mais aussi de plus en plus complexes et coûteux en calcul. Les observations, elles, sont de plus en plus abondantes grâce notamment aux satellites mais elles sont aussi très hétérogènes de par leur nature, leur densité et leur qualité. L'information nécessaire pour faire une bonne prévision est donc répartie dans les équations du modèle et dans les observations : il faut les utiliser au mieux. La formulation générale du problème de l'assimilation de données peut alors s'énoncer : « comment utiliser simultanément toutes les sources d'information pour obtenir la meilleure prévision ? ».

Plus précisément, l'assimilation de données consiste à déterminer la meilleure condition initiale possible pour le modèle à partir des observations que l'on connaît. On sait depuis les travaux de Lorenz en 1963 [45] que l'atmosphère est chaotique et que la qualité d'une prévision dépend dans une large mesure de la qualité de la condition initiale utilisée. L'assimilation de données est donc une phase essentielle qui conditionne la qualité des prévisions et à laquelle les services de météorologie accordent de plus en plus d'importance et de temps de calcul : de négligeable au début, équivalente à une prévision à 24 heures actuellement et bientôt équivalente à une prévision à 10 jours.

Nous savons aussi que le coût en calcul de l'intégration d'un modèle atmosphérique ou océanique est de plus en plus grand. Et, on estime actuellement que le

coût d'une bonne assimilation de données se situe entre 10 et 100 fois le coût de la prévision. Pour la prochaine génération de modèles, cela nécessiterait une puissance de calcul de l'ordre de 10 Tflops. À l'heure actuelle, aucun calculateur n'est capable de fournir de telles performances mais il est raisonnable de penser que cela sera possible dans quelques années, en particulier grâce aux calculateurs parallèles à mémoire distribuée. C'est la raison pour laquelle nous nous intéressons dès à présent à la parallélisation de ce problème.

Mais la programmation des machines parallèles reste encore un processus compliqué et on ne connaît pas de méthode générale pour paralléliser de manière optimale un algorithme donné. La difficulté est donc de développer des algorithmes numériques qui s'adaptent bien au calcul parallèle. Parmi les méthodes d'assimilation de données existantes, nous nous intéresserons à la méthode variationnelle. Cela nous conduira à étudier la parallélisation d'algorithmes numériques d'optimisation assez généraux.

En ce qui concerne l'aspect pratique, nous avons effectué une grande partie de ce travail sur un réseau de stations de travail avant d'avoir à notre disposition un IBM SP1 à l'Institut IMAG et un Cray T3D au CENG. Nous nous placerons dans cette étude dans le cas d'une machine dédiée à notre application, c'est-à-dire que nous ne tiendrons pas compte des aspects de partage des ressources avec d'autres applications éventuelles. Cela représente bien ce qui se passe dans les services de prévision opérationnelle où l'on recherche avant tout des performances maximales.

Le but de cette thèse est de montrer comment l'assimilation variationnelle peut être utilisée pour réaliser des algorithmes parallèles efficaces et pour coupler des modèles.

Après une présentation des outils du parallélisme et des bases de l'assimilation de données, nous tenterons de répondre au problème de la parallélisation de l'assimilation de données. Nous l'appliquerons au modèle de Shallow water et donnerons quelques indications pour l'application à ARPS. Plus précisément, le plan que nous suivrons dans ce document sera le suivant :

Dans le premier chapitre, nous présenterons le calcul parallèle en général, notamment les notions qui seront utilisées dans la suite. Nous introduirons tout d'abord les motivations qui ont conduit au calcul parallèle et les différentes formes que celui-ci peut prendre. Nous détaillerons ensuite les modèles de programmation des machines parallèles puis les environnements qui sont à la disposition du programmeur. Enfin, nous étudierons la manière dont on peut mesurer les performances d'un programme parallèle, ce qui nous permettra ensuite d'évaluer nos algorithmes.

Le deuxième chapitre sera consacré à une présentation plus approfondie de la programmation par échange de messages ainsi que des deux bibliothèques les plus

utilisées dans ce cadre : PVM et MPI, que nous utiliserons pour expérimenter les algorithmes que nous proposons.

Dans le troisième chapitre, nous introduirons le problème de l'assimilation des données en météorologie et en océanographie. Nous présenterons rapidement les différents algorithmes existants puis nous nous attacherons plus en détail à l'assimilation variationnelle et à l'écriture et l'utilisation des modèles adjoints. Nous étendrons la méthodologie de l'écriture des modèles adjoints au cas où le modèle direct est parallèle avec échanges de messages explicites.

Le quatrième chapitre présentera les différentes approches possibles *a priori* pour paralléliser la résolution du problème de l'assimilation de données. Cela peut se faire à plusieurs niveaux : au niveau des modèles météorologiques direct et adjoints, au niveau de l'algorithme d'optimisation ou enfin au niveau du problème lui-même. Cela nous conduira à transformer un problème séquentiel d'optimisation sans contraintes en un ensemble de problèmes d'optimisation relativement indépendants qui pourront être résolus en parallèle. On étudiera plusieurs variantes de ces trois approches très générales et leur utilité dans le cadre du problème de l'assimilation de données.

Dans le cinquième chapitre, nous présenterons l'application des méthodes de parallélisation du chapitre précédent au modèle de Shallow water. Nous commencerons par une courte présentation de ce modèle puis nous étudierons sa parallélisation ainsi que celle de son code adjoint et les performances obtenues. Nous présenterons ensuite plusieurs algorithmes parallèles d'assimilation de données et comparerons leurs performances.

Dans le dernier chapitre, nous présenterons le modèle météorologique ARPS (Advanced Regional Prediction System) qui est un modèle régional conçu pour la prévision des tornades. Nous présenterons une parallélisation de ce modèle écrite en collaboration avec K. Johnson (SCRI, Florida State University) et présenterons les performances obtenues.

Nous terminerons par quelques remarques générales que l'on peut retenir de ce travail et quelques perspectives.

Chapitre 1

Le parallélisme

Dans ce chapitre, nous présentons le calcul parallèle en général, notamment les notions qui seront utilisées dans la suite de cette thèse. Nous introduisons tout d'abord les motivations qui ont conduit au calcul parallèle et les différentes formes que celui-ci peut prendre. Ensuite nous présentons la manière dont sont mises en œuvre les communications dans les machines parallèles. Nous détaillerons ensuite les modèles de programmation des machines parallèles puis les environnements qui sont à la disposition du programmeur. Enfin, nous étudierons la manière dont on peut mesurer les performances d'un programme parallèle.

1.1 Introduction

Le principe de base qui est à l'origine du calcul parallèle est à la fois simple et ancien : le travail avance plus vite à plusieurs que seul. C'est une idée qui est déjà exploitée dans la plupart des activités humaines par des entreprises regroupant de plus en plus d'hommes et de moyens.

Cependant, elle n'a été appliquée que récemment au domaine des ordinateurs, premièrement car le besoin s'en est fait plus pressant, deuxièmement car les technologies nécessaires (réseaux) ont connu des progrès significatifs dans les vingt dernières années. En effet, jusqu'à ces dernières années, la technologie des microprocesseurs a fait des progrès tels que leur vitesse de calcul a crû de manière exponentielle. On constate sur le tableau 1.1 un ralentissement de ces progrès depuis quelques années. En effet, depuis 1976, c'est-à-dire en 20 ans, le temps de cycle des processeurs les plus rapides du marché a diminué d'un facteur 3 seulement alors que, pendant les 30 ans qui ont précédé, on a gagné un facteur 10^5 à 10^6. Cette limitation n'est plus seulement technologique, elle est aussi liée à des facteurs plus fondamentaux. Pour améliorer encore le temps d'un cycle d'un processeur, il faut gagner sur le temps de commutation des circuits mais aussi sur le temps de propagation des signaux entre ces circuits qui devient un facteur limitant bien que les signaux électriques se déplacent à la vitesse de la lumière. Il faut donc rapprocher les circuits, c'est-à-dire miniaturiser encore les composants. Or la technologie actuelle permet de fabriquer

des composants tels que si l'on diminue encore leur taille (la grandeur caractéristique dans les composants actuels est de l'ordre de $10^{-1}\mu$m), il faudra tenir compte de l'effet tunnel qui permet aux électrons de « sauter » d'un conducteur à un autre si ceux-ci sont trop proches ($10^{-3}\mu$m) (voir [48]).

Néanmoins, si l'on regarde la dernière colonne du tableau 1.1, on s'aperçoit que malgré la stabilité du temps de cycle des processeurs, leurs performances augmentent : les constructeurs utilisent déjà à l'intérieur des processeurs certaines formes de parallélisme.

Processeur	Année	Temps de cycle	Performances[1]
Mark I	1944	6 s	$<$ 1 flops
ENIAC[2]	1946	3 ms	300 flops
DEC PDP-1	1961	5μs	200 kflops
IBM 360	1964	250 ns	4 Mflops
Cray-1	1976	12.5 ns	80 Mflops
Cray X-MP	1982	9.5 ns	210 Mflops
Cray Y-MP	1988	6 ns	333 Mflops
Cray C90	1993	4.2 ns	952 Mflops

TAB. 1.1 - *Performances de divers processeurs.*

Un autre fait important qui a favorisé le développement du parallélisme dans les ordinateurs est économique : il revient moins cher de fabriquer un calculateur à partir de plusieurs processeurs de grande série que de concevoir un processeur spécifique très rapide. Le prix de revient au Mflops[3] peut varier dans des proportions de 1 à 10. En effet, d'après G. Bell (préface de [34]), le prix de revient d'un Mflops sur un Cray C90 est d'environ 2000 dollars alors qu'il est de 200 dollars seulement sur une station de travail. Cela explique l'intérêt de nombreux centres de calcul pour les ordinateurs parallèles, même si ces prix sont évalués pour des puissances théoriques qui ne seront pas atteintes dans les applications réelles.

1.2 Les différents types de parallélisme

1.2.1 Processeurs vectoriels

Pour continuer l'analogie avec l'activité humaine, la première méthode de travail à plusieurs est le travail dit « à la chaîne ». C'est le principe de fonctionnement des machines **vectorielles**. Dans ce type de machines, les différentes tâches effectuées simultanément le sont à l'intérieur d'un même processeur par des circuits spécialisés. Lorsque la phase d'amorçage de la chaîne (appelée **pipeline** en informatique) est

1. Vitesse maximum théorique en nombre d'opérations flottantes par seconde.
2. Le temps donné ici est en fait le temps de calcul d'une multiplication.
3. Million d'opérations flottantes par seconde, voir paragraphe 1.6.1.

terminée, l'intervalle de temps entre la sortie de deux résultats successifs est beaucoup plus court que le temps total de calcul complet d'un résultat. Plus précisément, il est égal au temps que prend une étape du calcul (voir figure 1.1), c'est-à-dire généralement un cycle d'horloge. Une opération flottante (addition ou multiplication par exemple) se décompose en quelques étapes (en général 4 ou 5). On gagne au maximum un facteur égal à ce nombre d'étapes, ce qui est encore limité.

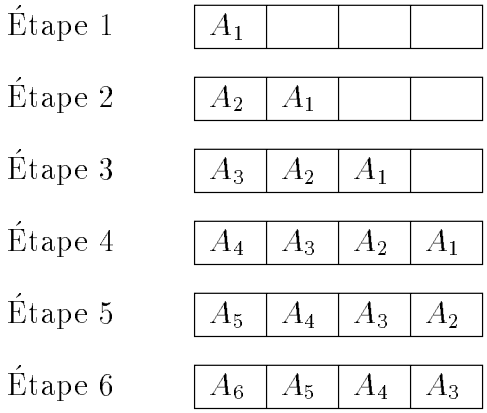


FIG. 1.1 - *Principe du pipeline : il faut 4 étapes pour effectuer le premier calcul (A_1), ensuite on obtient un nouveau résultat à chaque étape.*

Pour obtenir les meilleures performances possibles avec ce type de processeur, il est important de veiller à ce que le pipeline soit alimenté en continu. En effet, un calcul isolé coûte le temps de toutes les étapes du pipeline alors qu'un calcul en milieu de série coûte le temps d'une étape. La technique dite de « vectorisation » qui permet cela est maintenant connue et les compilateurs exploitent bien les pipelines de manière automatique et transparente à l'utilisateur.

1.2.2 Ordinateurs à mémoire partagée

Pour gagner encore en performance sont apparues les premières machines à plusieurs processeurs. Ce phénomène a tout d'abord touché les super-calculateurs (Cray-2 ou Cray C90 par exemple) et s'étend maintenant aux stations de travail (SGI, 2 ou 4 processeurs). Ces machines qui possèdent jusqu'à 16 ou 32 processeurs vectoriels très rapides et une mémoire commune peuvent ainsi atteindre des vitesses de calcul de l'ordre de 10 Gflops. Les problèmes technologiques posés par ce type d'architecture viennent de la mémoire et du bus qui relie les processeurs à cette mémoire qui doivent être très rapides pour satisfaire toutes les demandes.

Il faut également gérer la cohérence de cette mémoire (il ne faut pas que plusieurs processeurs modifient une valeur en même temps). Cette tâche peut être prise en charge par le système d'exploitation ou laissée à l'utilisateur, mais dans tous les cas

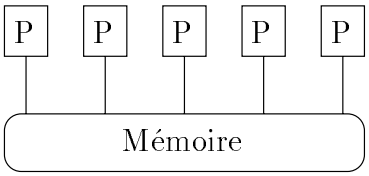


FIG. 1.2 - *Ordinateur à mémoire partagée.*

cela devient difficile à gérer et trop coûteux en temps au-delà de 16 processeurs. La programmation de ces machines oblige aussi à prendre garde à la validité des données au moment de les utiliser. Toutes les opérations précédentes ont-elles été effectuées (par le processeur courant ou un autre) au moment d'utiliser une donnée? Il faut gérer la synchronisation des processus. Ce type de parallélisme peut être exploité de manière automatique (*multitasking* sur Cray par exemple) en obtenant des performances relativement bonnes.

1.2.3 Ordinateurs à mémoire distribuée

Des constructeurs ont alors proposé une autre solution : associer une mémoire propre à chaque processeur. Les processeurs deviennent totalement indépendants et sont connectés par un réseau (voir figure 1.3) qui leur permet d'échanger l'information nécessaire pour coordonner leur travail. On appelle ces machines les ordinateurs à mémoire distribuée. Les premières machines de ce type ont été construites avec un grand nombre de processeurs peu puissants (comme par exemple la CM2 qui comportait jusqu'à 65536 processeurs 1 bit). Grâce aux progrès réalisés dans la technologie des réseaux d'interconnexion, on voit apparaître des machines basées sur des processeurs de grande série et de plus en plus rapides dont la puissance crête varie entre 50 et 200 Mflops (processeurs issus des stations de travail) mais moins nombreux, c'est le cas notamment des IBM SP1 et SP2 et du Cray T3D. Ces processeurs semblent fournir un bon compromis pour obtenir une grande puissance de calcul tout en gardant un rapport vitesse de calcul / vitesse de communication raisonnable. Il est important de conserver un équilibre entre les performances des processeurs et du réseau d'une machine. En effet, il est inutile d'utiliser des processeurs très performants si le réseau est lent car ils seront limités par la vitesse de transfert des données sur ce réseau. L'utilisation de processeurs de grande série tels que ceux qui équipent les stations de travail permet de réduire les coûts. Actuellement, les constructeurs proposent des machines ayant jusqu'à quelques centaines de processeurs de ce type.

Théoriquement, une machine possédant 500 processeurs de 100 Mflops chacun a une vitesse crête de 50 Gflops. En réalité ce chiffre est loin d'être atteint. En effet, le temps de communication pendant l'exécution d'un programme parallèle est du temps perdu, il faudra donc toujours chercher à le réduire le plus possible. Il faut donc **revoir entièrement les méthodes numériques** utilisées et en mettre au

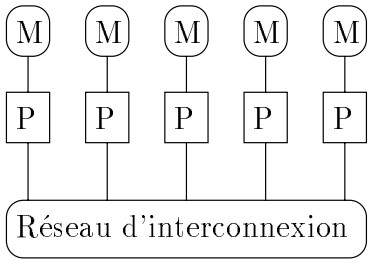


FIG. 1.3 - *Ordinateur à mémoire distribuée.*

point de nouvelles qui comportent le minimum de dépendance et de communications entre les sous-parties des calculs à effectuer.

1.3 Les communications

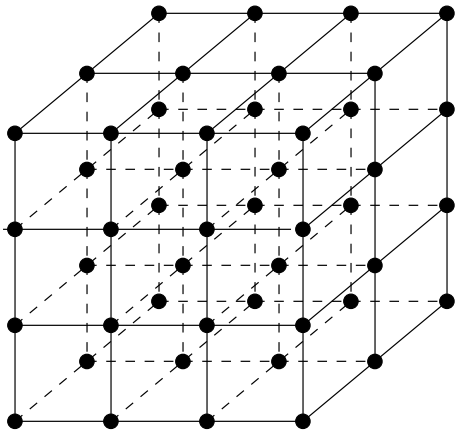
Dans cette partie, nous allons détailler quelques notions sur les communications dans les réseaux de processeurs qui nous seront utiles dans la suite. Nous allons commencer par nous intéresser au niveau physique, c'est-à-dire aux réseaux de communication puis au niveau logique, c'est-à-dire aux schémas de communication mis en œuvre sur ces réseaux.

1.3.1 Réseaux

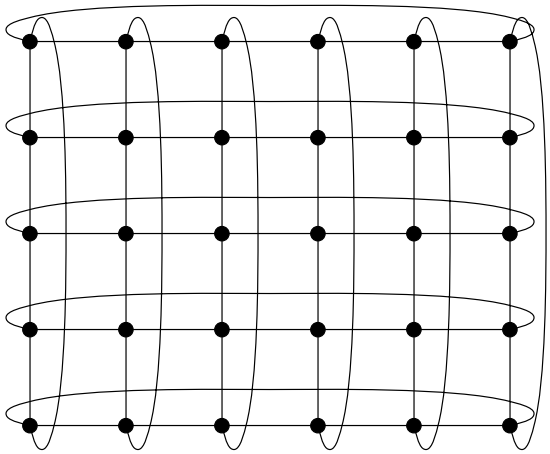
Une machine parallèle à mémoire distribuée comprend deux types de composants : des processeurs et un réseau leur permettant de communiquer. Les constructeurs ont proposé une très grande variété de réseaux sur leurs machines. Les principales topologies utilisées ont été :

- les hypercubes (CM2, iPSC860),
- les grilles (Paragon),
- les tores (T3D, Maspar),
- les *fat-tree* (CM5),
- les réseaux multi-étages (SP1, CS2).

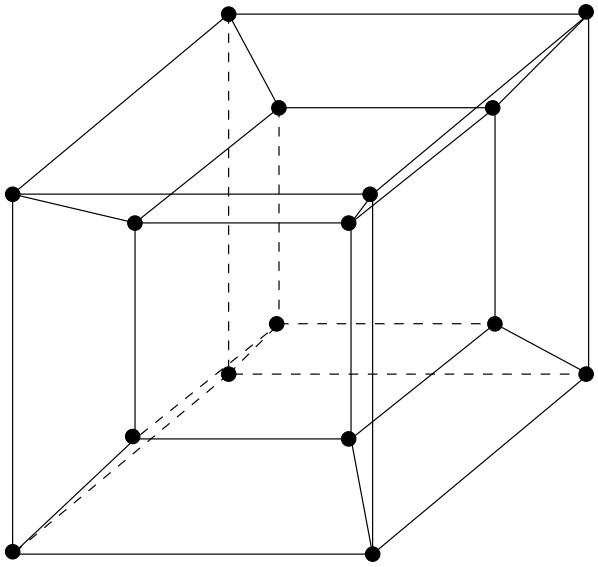
La figure 1.4 montre quelques exemples de ces réseaux. L'hypercube fut très à la mode dans les années 80 mais a tendance à disparaître car son degré (nombre de connexions par processeur) augmente rapidement avec sa taille, ce qui pose des problèmes technologiques de réalisation. Les principaux types de configurations toujours utilisés actuellement sont les grilles et tores ainsi que les réseaux multi-étages.



Grille de dimension 3



Tore de dimension 2



Hypercube de dimension 4

FIG. 1.4 - *Quelques topologies de machines parallèles.*

Nous utiliserons au-dessus de ces réseaux et de manière plus ou moins transparente à l'utilisateur des schémas de communication logiques qui sont présentés dans le paragraphe suivant.

1.3.2 Schémas de communication

Selon le réseau dont est constituée une machine parallèle, chaque processeur est connecté avec un nombre plus ou moins important d'autres processeurs de la machine. Bien sûr, cela ne suffit en général pas et chaque processeur doit pouvoir

échanger des données avec tous les autres. Pour cela, nous utilisons des algorithmes de **routage** qui peuvent être réalisés par matériel ou par logiciel. Nous utiliserons des schémas de communication de haut niveau concernant un grand nombre de processeurs et/ou des processeurs très distants dont les principaux types sont :

Communication point à point : communication d'un processeur à un autre quelle que soit leur position relative dans le réseau.

Diffusion : communication d'un processeur vers tous les autres.

Échange total : diffusion simultanée à partir de tous les processeurs.

Distribution : communication d'un processeur vers tous les autres, le message envoyé à chaque processeur étant différent.

Multi-distribution : distribution simultanée à partir de tous les processeurs.

Regroupement : tous les processeurs envoient un message à un processeur donné (c'est l'inverse de la distribution). On parle de combinaison lorsqu'il y a des calculs à chaque étape intermédiaire (comme la recherche du maximum ou le calcul de la somme d'un ensemble de valeurs).

De nombreux travaux de recherche ont lieu sur les algorithmes de routage efficaces pour ces opérations en fonction du type de réseau physique utilisé. Les machines et/ou bibliothèques de communication modernes comprennent des appels à ces opérations qui sont implémentées de manière efficace. Pour plus de détails sur ces algorithmes, on peut consulter le livre de l'école d'été Rumeur [57].

1.3.3 Performances des réseaux

Le temps nécessaire pour transmettre un message de longueur L est la somme :

- d'un temps d'initialisation β représentant le temps des initialisations et appels de procédures,
- d'un temps de transmission du message proportionnel à la taille du message, on notera τ_c le temps de transmission d'un octet.

Le temps de transmission du message est donc :

$$T = \beta + L\tau_c.$$

On caractérisera donc les performances d'un réseau par deux paramètres : sa latence β et son débit τ_c ou sa bande passante $1/\tau_c$, on donne dans le tableau 1.2 (obtenu d'après [25] et [13]) les valeurs de ces paramètres pour quelques machines représentatives. Il convient de prendre ces chiffres avec beaucoup de prudence car, comme on peut le voir dans [13], ils varient beaucoup selon les couches logicielles utilisées. Dans l'état actuel de la technologie, les réseaux sont le point faible des machines parallèles : ils ne permettent pas d'alimenter les processeurs aussi vite qu'ils ne calculent (sauf dans le cas du Cray T3D). Il sera donc très important de minimiser le temps de communication dans les programmes parallèles.

Machine	Latence (μs)	Débit (Mo/s)
CM 5	82	2.3
Paragon	121	14.3
SP2	40	9.1
T3D	119	28.5
Ethernet	1500	0.2

TAB. 1.2 - *Performances mesurées de quelques réseaux de communication.*

1.3.4 Minimisation du temps de communication

Nous avons vu que, sur une machine parallèle à mémoire distribuée, les communications entre les processeurs sont nécessaires, mais qu'elles prennent du temps. Il faudra donc chercher à les minimiser, ou tout au moins à minimiser leur impact sur le temps total d'exécution. Pour cela on dispose de deux méthodes : soit on minimise le nombre et le volume des communications par un découpage astucieux du programme, soit on masque le temps de communication, c'est-à-dire que l'on écrit le programme de telle façon que les échanges de données aient lieu en même temps que les calculs.

Nous allons illustrer ces notions sur un exemple simple : considérons un calcul itératif explicite de type différences finies sur une grille bidimensionnelle et utilisant un schéma à cinq points. On suppose que la grille est de taille $N \times M$, qu'elle est repérée par les indices i et j et qu'elle est distribuée sur P processeurs. On notera τ_a le temps de calcul en chaque point de la grille, le temps de communication pour un message de taille L sera $\beta + L\tau_c$. On supposera dans ce paragraphe que toutes les opérations effectuées « tombent juste », cela ne change rien à la méthode et permet d'alléger les notations.

Nous supposerons dans un premier temps que les données sont distribuées entre les processeurs selon un algorithme «modulo», c'est-à-dire que la donnée $[i, j]$ est attribuée au processeur $(i + j) \bmod P$. Dans le cas général, pour effectuer un calcul, il faudra communiquer avec les processeurs qui gèrent les quatre points voisins du point courant. Le temps total de calcul d'une itération sera donc :

$$T = \frac{N \times M}{P} \left(\tau_a + 4 \times (\beta + \tau_c)\right).$$

On peut améliorer ce temps en changeant la distribution des données sur les processeurs. Affectons à chaque processeur une sous-grille de taille $(N/\sqrt{P}) \times (M/\sqrt{P})$. Quel est le temps d'une itération avec cette distribution? Le temps de calcul est le même, par contre le temps de communication est meilleur. En effet, pour tous les points à l'intérieur de la sous-grille, aucune communication n'est nécessaire, pour les points sur les frontières, on a besoin d'une communication, ou de deux dans les

coins. Le temps total est donc :

$$T = \frac{N \times M}{P}\tau_a + 2 \times (N/\sqrt{P} + M/\sqrt{P}) \times (\beta + \tau_c).$$

De plus, toutes les communications sur un côté du domaine ont lieu avec le même processeur, on peut les regrouper en un seul message pour diminuer le temps de latence (voir figure 1.5). Le temps total est alors :

$$T = \frac{N \times M}{P}\tau_a + 4\beta + 2(N/\sqrt{P} + M/\sqrt{P})\tau_c.$$

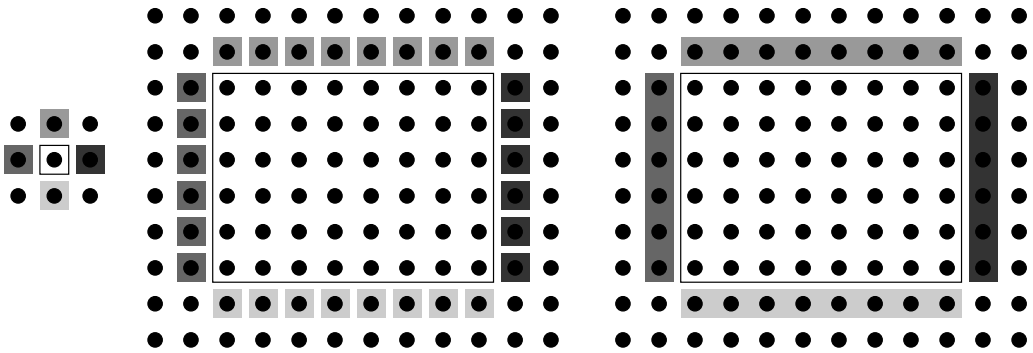


FIG. 1.5 - *Distribution de données et fractionnement des communications : à gauche, les points sont séparés, pour chacun d'entre eux quatre échanges de messages sont nécessaires; au centre, on groupe les points par sous-domaines, on élimine ainsi un grand nombre de communications à l'intérieur des sous-domaines; à droite, on groupe les communications entre deux sous-domaines dans un seul message.*

Mais il est possible de faire encore mieux ! Le procédé décrit étant itéré, on peut recouvrir les communications par du calcul. Pour cela, le procédé est le suivant : à une itération donnée, on commence par calculer les valeurs des points à la limite du domaine (dont les processeurs voisins auront besoin à l'étape suivante). On peut alors les envoyer sans attendre et on procède de la même façon sur tous les processeurs. On calcule ensuite les valeurs internes au domaine. Pendant ce calcul, les messages se propagent sur le réseau. À l'étape suivante, les données sont donc prêtes à être utilisées (voir figure 1.6). Sous réserve que le temps de calcul soit suffisant pour que les messages arrivent, le temps total d'une itération est égal au temps de calcul sur un domaine, c'est-à-dire :

$$T = \frac{N \times M}{P}\tau_a.$$

En pratique, l'évaluation du temps est un peu plus compliquée. Il faut tenir compte de l'architecture de la machine. Les temps qui sont donnés ici supposent que la machine dispose d'un mécanisme matériel d'envoi des messages qui ne perturbe pas le processeur de calcul. Il se peut que celui-ci soit interrompu si la machine ne dispose pas de coprocesseur de communication. On gagne alors seulement la partie due à l'attente des messages mais le principe reste valable.

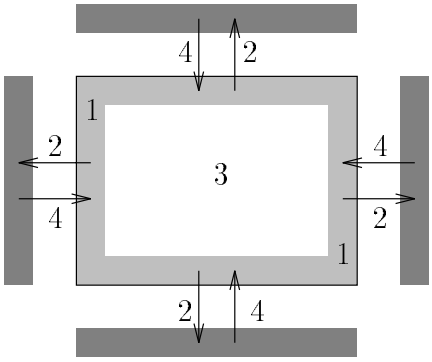


FIG. 1.6 - *Masquage des communications : on effectue le calcul sur les parties frontières des domaines (en grisé) puis on envoie les valeurs utiles aux autres processeurs; pendant que les messages transitent sur le réseau on effectue le calcul à l'intérieur des domaines et on reçoit les données des autres processeurs à la fin de l'étape de calcul.*

1.4 Les modèles de programmation

1.4.1 Parallélisme implicite ou explicite ?

Il existe plusieurs manières de programmer une machine parallèle. On peut utiliser un paralléliseur automatique, dans l'état actuel des connaissances cette solution est à éviter car si elle est pratique, elle n'est en général pas suffisamment efficace. Il faut donc encore écrire à la main des programmes explicitement parallèles.

Pour cela, il nous faut découper notre programme sous forme de tâches qui pourront s'effectuer simultanément sur différents processeurs et définir la manière dont ces tâches vont coopérer afin de réaliser le travail souhaité.

1.4.2 Parallélisme de données ou de contrôle ?

Pour réaliser ce découpage, on peut utiliser un modèle dit de «parallélisme de données». Dans ce modèle de programmation, le découpage du programme est guidé par les données qu'il manipule : on découpe ses données et on les distribue dans les mémoires liées aux différents processeurs. On effectue ensuite localement les traitements mettant ces données en jeu. Il est évident que certaines opérations nécessiteront des données résidant sur des processeurs distincts : il faudra alors communiquer des données par le réseau. La difficulté réside dans le choix du découpage initial qui induira plus ou moins de communications dans le déroulement du programme.

Le deuxième modèle de programme parallèle est appelé «parallélisme de contrôle». Dans ce modèle on découpe le programme selon les différentes tâches à effectuer. On ajoute à ce découpage une description de l'ordre dans lequel ces tâches doivent

s'enchaîner et on en déduit les mouvements de données nécessaires. Un avantage important de cette approche est qu'elle s'adapte mieux aux applications numériques irrégulières ainsi qu'à certaines applications autres que le calcul scientifique.

1.4.3 Expression du parallélisme

Quel que soit le modèle de programmation que l'on désire utiliser, il existe plusieurs manières de l'exprimer.

Il existe des langages spécifiques au parallélisme de données. Ils sont en général basés sur Fortran et sont surtout utilisés pour les grosses applications numériques, la parallélisation s'exprimant souvent au moyen de directives de compilation. Le principe est très simple, on distribue les tableaux de données (statiques en Fortran) entre les processeurs, le compilateur se chargeant de découper les boucles de calcul de manière appropriée et de placer les échanges de messages nécessaires. C'est la méthode utilisée dans CRAFT [18] et dans HPF (High Performance Fortran, [33]). L'usage de ces langages est relativement simple et rapide à mettre en œuvre, la génération automatique des échanges de messages n'est malheureusement pas encore aussi efficace qu'elle pourrait l'être et constitue un thème de recherche important.

Plusieurs équipes de recherche travaillent actuellement à l'élaboration d'environnements et de langages de programmation utilisant le modèle de parallélisme de contrôle [54]. Ce type de langage est le plus souvent basé sur C ou C++ et utilise des mécanismes tels que l'appel de procédures à distance (RPC). On envoie à un processeur un appel à une fonction ou procédure et ses arguments, on reçoit le résultat correspondant à la fin de son exécution.

Une autre solution est la programmation par échange de messages, c'est l'approche que nous utiliserons. Cette manière de programmer est la plus proche de ce qui se passe réellement, on écrit un ou des programmes séquentiels qui s'exécuteront sur les processeurs de la machine parallèle et qui comprennent des instructions d'envoi et de réception de données. On peut écrire de cette façon des programmes dans le modèle de parallélisme de données ou de parallélisme de contrôle ou même mixtes. Ce type de modèle est utilisé par de nombreuses bibliothèques portables ou spécifiques à une machine. Des exemples et des références sur ce type de programmation seront détaillés dans le chapitre 2.

En résumé, on peut dire que l'efficacité obtenue est pour le moment inversement proportionnelle à l'automatisme et au niveau de programmation utilisés. C'est là le principal défaut des machines parallèles à l'heure actuelle : il faut avoir des connaissances spécifiques pour en tirer de bonnes performances. Le faible prix de revient des heures de calcul sur les machines parallèles a pour contrepartie un coût de conception des codes élevé.

1.5 Environnements de programmation

Après avoir défini ce qu'est une machine parallèle et les modèles que l'on peut utiliser pour les programmer, détaillons le cycle de développement d'un code parallèle. Nous supposerons dans ce paragraphe que nous avons choisi un modèle de programmation et un langage pour l'exprimer. En fonction du modèle et du langage choisis, certains des points qui suivent pourront être plus ou moins automatisés. Néanmoins, il est important de connaître ces aspects pour concevoir un bon programme parallèle.

1.5.1 Placement, ordonnancement

La première étape de la conception d'un programme parallèle est de le décomposer en tâches. Ensuite, il faut placer et ordonnancer ces tâches, c'est-à-dire attribuer à chaque tâche du programme un processeur sur lequel elle va s'exécuter et déterminer l'ordre d'exécution des tâches sur chaque processeur. Un bon placement doit minimiser une fonction de coût qui caractérise la qualité de l'exécution d'un programme parallèle (le temps d'exécution par exemple) sur une machine donnée. Les deux principaux goulots d'étranglement sont la charge des processeurs et le coût des communications. En entrée de cette phase, on suppose que l'on a un graphe représentant le programme. Pour être complet, ce graphe doit contenir la durée des tâches et des communications ainsi que les contraintes de précédence des tâches entre elles. On dispose également d'un graphe qui représente la machine cible. Ce graphe contient deux paramètres importants qui sont la vitesse des processeurs et la vitesse des communications (toutes deux en fonction de la charge) ainsi que la topologie du réseau de communication.

Le problème de placement qui se pose alors est NP-complet pour un graphe quelconque, c'est-à-dire qu'il est impossible de le résoudre en un temps polynomial en fonction du nombre de tâches et du nombre de processeurs. Il faut donc recourir à des heuristiques pour le résoudre. Plusieurs types d'heuristiques existent (voir [31]) :

liste : on place les tâches les unes après les autres suivant un critère donné (la plus longue sur le processeur le moins chargé par exemple). Un processus une fois placé n'est plus remis en question.

itératif : partant d'un placement on le perturbe pour essayer d'obtenir un meilleur résultat (l'algorithme tabou est le plus performant actuellement).

regroupement : on regroupe les tâches en groupes (choix d'une granularité) afin de diminuer la taille du graphe puis on place ces groupes à l'aide d'une des méthodes précédentes.

Les placements étant obtenus par des heuristiques, il est nécessaire de les évaluer, ce qui augmente le coût de ces méthodes.

1.5.2 Mise au point

La mise au point d'un programme parallèle est un processus coûteux en temps. Elle se compose de deux aspects : la mise au point séquentielle d'une tâche donnée (on utilise des outils UNIX classiques) et la mise au point de la partie communications. Cette deuxième phase est difficile car il n'existe pas d'outils bien adaptés. On utilise la plupart du temps un débogueur classique par tâche à déboguer, ce qui peut vite devenir inextricable.

La mise au point d'un programme parallèle est compliquée par l'interaction entre les tâches qui rend son comportement indéterministe. Un algorithme numérique séquentiel est déterministe, c'est-à-dire que, pour un jeu de données fixé, la suite des instructions exécutées lors de différentes exécutions sera toujours la même. Ce n'est pas le cas pour un programme parallèle. Considérons par exemple le cas de deux processus d'un programme parallèle qui envoient chacun un message à un troisième processus. Ce processus est programmé pour attendre le premier message qui arrive et poursuivre son exécution en fonction de ce message. Selon la charge des processeurs qui exécutent les processus émetteurs et la charge du réseau de communication, les messages arriveront dans un ordre différent, ce qui modifie la suite d'instructions exécutées. Le comportement est donc **indéterministe**.

L'indéterminisme implique qu'une erreur survenant lors d'une exécution peut ne pas se reproduire dans une autre. La difficulté de mise au point est donc beaucoup plus grande. Un autre facteur rend la mise au point encore plus difficile : les procédés de mise au point classiques perturbent l'exécution et rajoutent des retards (points d'arrêt), le programme peut donc avoir un comportement encore différent. L'observation modifie le comportement du système : ce phénomène est connu en physique sous le nom de *probe effect*, en informatique on dit aussi que la prise de traces est intrusive. La mise au point rajoute de l'indéterminisme et peut masquer des erreurs. Il faut disposer de mécanismes de **ré-exécution** déterministe pour pallier à cet inconvénient.

La ré-exécution consiste à mémoriser une exécution de manière à pouvoir la «rejouer» de façon déterministe (identique à l'exécution originale) autant de fois que nécessaire dans un environnement de débogage. Il faut pour cela «enregistrer» l'exécution originale et construire un ordre total sur tous les événements du programme, c'est ce que l'on appelle la **prise de traces**.

1.5.3 Traces

Le traçage d'un programme consiste à enregistrer ce qui se passe (identificateur d'événement), où (numéro de ligne) et quand (date) il a lieu ainsi que les circonstances de cet événement.

La prise de traces dans un programme parallèle est difficile à cause de deux facteurs : l'indéterminisme et le grand nombre possible de tâches qui multiplie d'autant le volume des données à recueillir. La méthode la plus courante consiste à ajouter des instructions de collecte d'information à des endroits judicieusement placés dans le code. Souvent on instrumente la bibliothèque de communication. Le problème posé est que cela modifie le temps d'exécution des tâches et donc ajoute de l'indéterminisme. Une autre technique possible est la collecte matérielle de traces. Les problèmes sont alors le coût, la non portabilité et le peu de flexibilité. Des travaux de recherche tentent de mettre au point des outils de prise de traces qui perturbent le moins possible l'exécution, ils sont dit peu **intrusifs**.

Dans un deuxième temps. il faut réorganiser les informations collectées localement sur chaque processeur. Cela pose des problèmes de datation si l'on ne dispose pas d'horloge globale. Le regroupement peut lui aussi perturber l'exécution en chargeant le réseau s'il a lieu pendant le déroulement de l'application, mais l'application est également perturbée par la diminution de l'espace mémoire disponible si les données sont conservées sur place.

L'introduction des instructions de traces peut aussi perturber le programme en empêchant le compilateur d'optimiser certaines parties du code (échange de l'ordre de boucles, *inlining* de procédures, modification de l'allocation des registres...). Dans un programme parallèle, cela peut même encore se compliquer : en HPF par exemple, le code compilé est très différent du code source, que tracer? L'ajout d'une instruction de traçage peut donner un code compilé complètement différent. Comment relier un événement du code binaire au code source? Il faudrait coupler la compilation à la prise de traces. Il n'existe pas actuellement de solution à ce problème.

Il faut trouver le bon compromis entre l'intrusion due à la prise de traces et la quantité d'information utile, ce compromis varie en fonction de ce que l'on recherche. Il faudra sélectionner des informations pertinentes pour le débogage ou l'analyse de performances. Enfin, il est nécessaire de filtrer et de réorganiser les données récoltées et de les présenter de manière exploitable par l'utilisateur.

1.5.4 Visualisation

Pour ne pas noyer l'utilisateur sous la quantité de données fournies par un traceur, la solution actuelle est de les représenter sous forme graphique. Nous présentons dans ce paragraphe deux manières de représenter graphiquement le déroulement d'une exécution d'un programme parallèle : le diagramme de Gantt et le diagramme espace-temps. Il existe d'autres façons de présenter le comportement d'un programme parallèle, chaque manière de faire possède ses avantages et ses inconvénients. De nombreux travaux de recherche sont en cours sur ce sujet car une exécution parallèle comporte de très nombreux paramètres qu'il n'est pas aisé d'exploiter.

Sur un diagramme de Gantt (voir figure 1.7), on représente l'activité des processeurs. Chaque processeur est représenté par une bande horizontale. Les parties sombres de la bande représentent les temps d'activité du processeur, les parties blanches les temps d'inactivité (attente de message par exemple) et les parties grisées les temps d'activité liés à la communication (surcoût constitué des temps de préparation des données et d'initialisation de la communication). On peut donc juger de la qualité de la parallélisation d'une application par la fraction de surface noire sur un tel diagramme et par sa répartition.

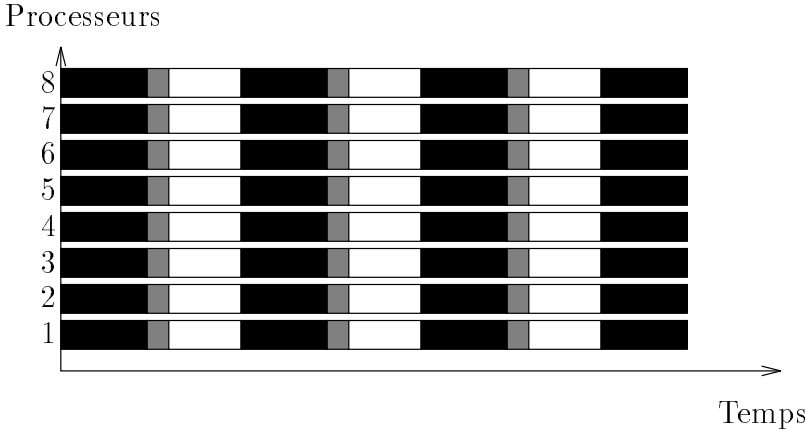


FIG. 1.7 - *Diagramme de Gantt.*

Sur un diagramme espace-temps (voir figure 1.8), on représente chaque processeur par une ligne horizontale, mais on représente aussi les communications entre les processeurs par des lignes obliques qui joignent les lignes représentant les processeurs qui communiquent. Une coupure dans une ligne horizontale correspond à un temps d'inactivité du processeur. Le principal problème lié à ce type de représentation est qu'il devient rapidement illisible dans les applications qui comportent beaucoup de communications.

Tous les graphiques présentés dans la suite de cette thèse ont été réalisés avec le logiciel de prise de traces TAPE [46] et le logiciel de visualisation Paragraph [52].

1.5.5 Répartition dynamique de la charge

Dans beaucoup de programmes parallèles, il est impossible de connaître *a priori* le temps d'exécution de certaines tâches. C'est le cas lorsque l'on utilise un maillage adaptatif ou lorsqu'une méthode itérative doit vérifier un critère de convergence par exemple. Il faut alors en cours d'exécution décider quelle est la meilleure façon de répartir les tâches entre les processeurs, on appelle ce procédé la **répartition**

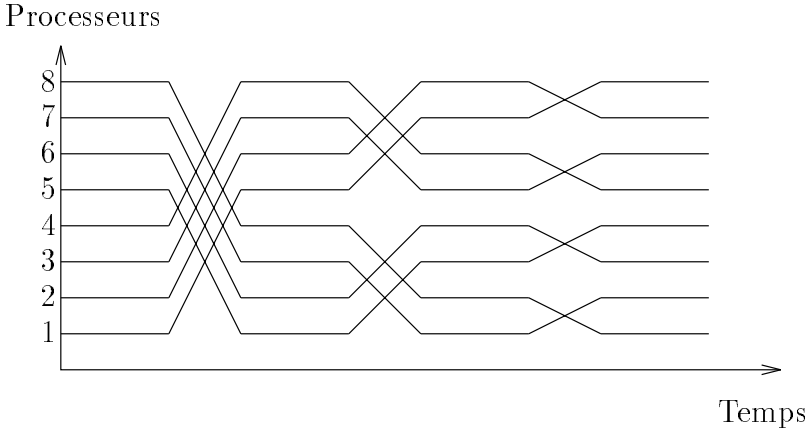


FIG. 1.8 - *Diagramme espace-temps.*

dynamique de la charge. On peut par le même genre de mécanisme prendre en compte la charge du réseau et/ou des processeurs qui sont eux-aussi imprévisibles dans le cas d'une machine multi-utilisateurs.

Les difficultés résident principalement dans la quantification de la charge d'un processeur et dans l'évaluation du temps qu'il reste pour terminer une tâche. Les algorithmes d'équilibrage de la charge sont également coûteux car ils nécessitent beaucoup de communications. La répartition dynamique de charge peut être prise en charge par le système d'exploitation, dans l'état actuel des connaissances, c'est en général le programmeur qui le fait dans son application.

1.6 Les performances

1.6.1 Les unités de mesure

Il existe de très nombreuses manières de mesurer les performances d'un ordinateur. Pour les applications qui nous intéressent ici, ce qui nous importe est la vitesse de calcul sur des variables réelles (simple ou double précision). On utilisera donc comme unité le **flops**, c'est-à-dire le nombre d'Opérations FLottantes Par Seconde, ou plus souvent ses multiples :

Mflops ou mégaflops : un million (10^6) d'opérations flottantes par seconde.

Gflops ou gigaflops : un milliard (10^9) d'opérations flottantes par seconde.

Tflops ou téraflops : 10^{12} opérations flottantes par seconde.

Pour donner un ordre de grandeur, disons que les vitesses actuelles de calcul des grosses stations de travail se situent entre 100 et 200 Mflops et que les supercalculateurs actuels les plus rapides atteignent des vitesses de quelques dizaines de

Gflops. Les constructeurs de ces machines promettent pour la fin de la décennie des machines capables de fournir une puissance de 1 Tflops.

Il convient de faire une distinction entre la puissance crête d'une machine et sa puissance dite soutenue. La puissance crête est la vitesse maximale théorique de la machine. En pratique celle-ci n'est jamais atteinte et ce pour plusieurs raisons : les applications contiennent des instructions autres que les opérations de calcul, les pipelines des machines ne sont pas alimentés de manière continue car le transfert des données depuis la mémoire est plus lent que le calcul. La puissance soutenue représente la vitesse maximale que l'on peut tirer d'une machine en application. Remarquons qu'elle n'est pas définie de manière précise mais qu'elle donne une idée de ce que l'on peut espérer.

Pour mesurer les performances d'un programme parallèle, on a besoin d'une autre notion que l'on peut appeler accélération ou efficacité. L'accélération est le rapport du temps d'exécution d'un programme séquentiel sur le temps d'exécution de la même application en parallèle :

$$A_n = \frac{T_{seq}}{T_n}$$

où A_n est l'accélération pour n processeurs, T_{seq} est le temps d'exécution du programme séquentiel et T_n le temps d'exécution sur n processeurs. L'efficacité est l'accélération divisée par le nombre de processeurs, on l'exprime en général sous forme de pourcentage :

$$E_n = \frac{A_n}{n} = \frac{T_{seq}}{n \times T_n}.$$

Ce nombre représente un pourcentage moyen d'utilisation des processeurs par rapport à une parallélisation qui serait parfaite.

1.6.2 La loi d'Amdahl

Il existe une borne théorique de l'accélération que l'on peut atteindre pour une application donnée sur un nombre de processeurs donné. Dans toutes les applications parallèles, il existe des parties qui ne sont pas parallélisables (par exemple l'initialisation de paramètres de calcul). Soit s la fraction du code (en temps d'exécution) qui n'est pas parallélisable. Le temps d'exécution minimum possible sur n processeurs est alors :

$$T_n = T_{seq} \times \left(s + \frac{1-s}{n} \right),$$

ce qui correspond à une accélération de :

$$A_n = \frac{n}{(1-s) + s \times n}$$

et à une efficacité maximum de :

$$E_n = \frac{1}{(1-s) + s \times n} .$$

La figure 1.9 nous montre la plus grande accélération que l'on pourrait obtenir si on avait une infinité de processeurs ainsi qu'avec 32 et 128 processeurs (cas des machines que nous utiliserons). La figure 1.10 nous montre l'efficacité théorique d'un programme parallélisé de 90% à 100% sur une machine à 32, 128 ou 1024 nœuds. On constate sur ces deux figures que plus la machine a de processeurs et plus la parallélisation doit être complète pour obtenir une bonne efficacité.

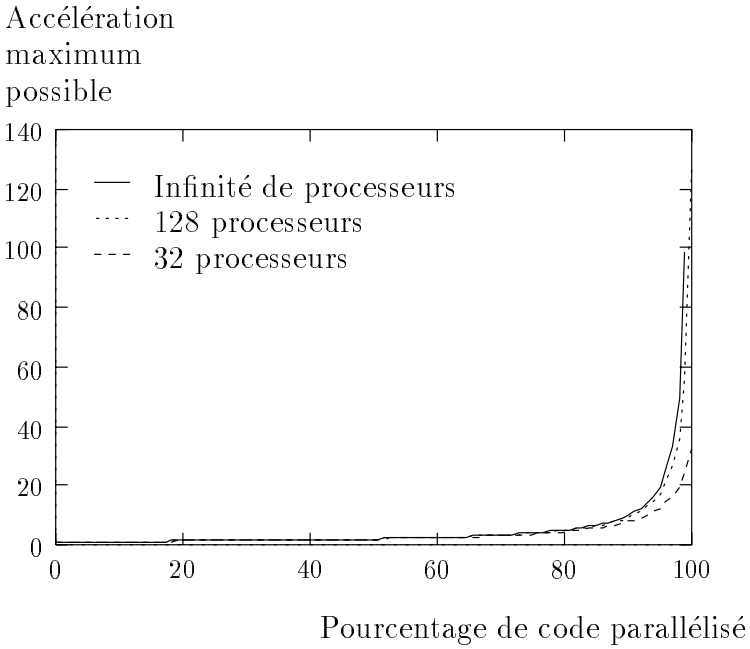


FIG. 1.9 - *Loi d'Amdahl: accélération théorique.*

On peut retenir de la loi d'Amdahl que si l'on veut accélérer de manière significative une application en utilisant une machine parallèle, il faut la paralléliser à fond. Paralléliser la ou les routines qui consomment le plus de temps ne suffit pas. Par exemple, paralléliser une portion du code qui consomme 80% du temps CPU sur une machine à 128 processeurs donnera une accélération maximale possible inférieure à 5 qui n'est pas satisfaisante. Pour atteindre une efficacité de 50% sur une machine à 32 nœuds, il faudra paralléliser 96% du code. Ce pourcentage est de 99% sur une machine à 128 processeurs.

Il existe deux manières d'interpréter la loi d'Amdahl. Elles diffèrent par la façon de mesurer le temps séquentiel. La première consiste à mesurer le temps d'exécution du même algorithme que celui utilisé en parallèle, l'autre consiste à prendre le temps

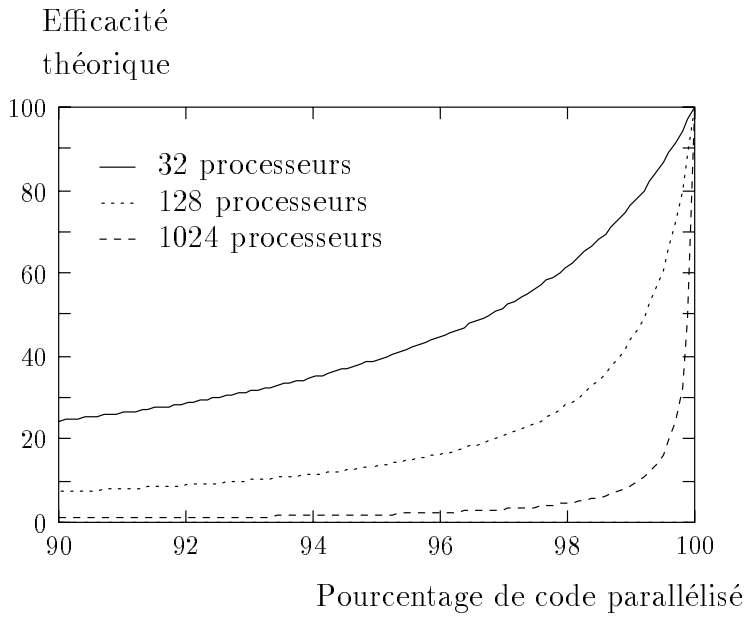


FIG. 1.10 - *Loi d'Amdahl: efficacité théorique.*

du meilleur algorithme séquentiel. Le problème réside alors dans la définition du meilleur algorithme séquentiel : il n'est pas toujours évident de savoir quel algorithme est le meilleur, cela peut varier en fonction des données et de l'implémentation, il se peut aussi que le meilleur algorithme du moment ne le soit plus dans un an. Il n'y a pas de bonne réponse, le tout est de donner clairement la référence que l'on utilise et d'en tenir compte au moment de tirer les conclusions d'une expérience.

1.6.3 La loi de Gustafson

Si l'on considère les bornes théoriques que nous donne la loi d'Amdahl, on ne peut être que très pessimiste quant à l'avenir du calcul parallèle. En fait, on peut avoir du problème une vision un peu différente comme celle de Gustafson [32].

En effet, lorsque l'on utilise un calculateur parallèle, la taille du problème et le nombre de processeurs que l'on utilise sont rarement indépendants. Les ingénieurs et chercheurs qui utilisent ce genre de calculateurs augmentent généralement la taille du problème le plus possible en fonction des moyens informatiques dont ils disposent. Selon leurs intérêts, les utilisateurs augmentent la finesse de leur grille de discrétisation ou le nombre de pas de temps ou choisissent une méthode de résolution plus précise pour avoir une meilleure réponse avec le même temps d'exécution. On constate aussi que la plupart du temps, la partie séquentielle du code ne change pas quand la taille du problème augmente alors que la partie parallèle croît.

Considérons un programme qui s'exécute en un temps T_n sur n processeurs et

soit s la fraction du code non parallélisable. Alors, le temps d'exécution de ce code sur un seul processeur sera égal au temps d'exécution de la partie non parallélisable plus n fois le temps d'exécution de la partie parallèle, soit :

$$T_1 = s \times T_n + n \times (1 - s) \times T_n.$$

On en déduit donc une nouvelle expression de l'accélération :

$$A_n = s + n \times (1 - s)$$

et de l'efficacité :

$$E_n = 1 - s + \frac{s}{n}\,.$$

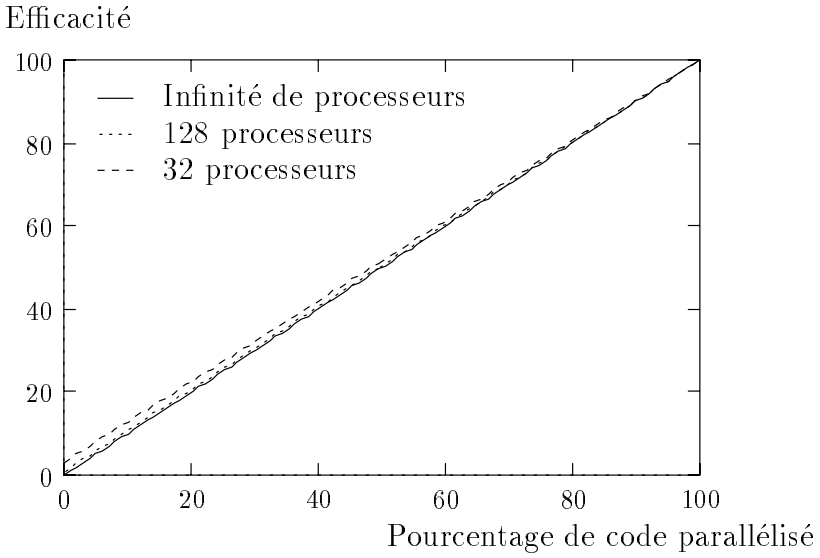


FIG. 1.11 - *Loi de Gustafson : efficacité théorique.*

Cette approche rejoint une autre notion que l'on trouve dans les ouvrages traitant de calcul parallèle : l'extensibilité (*scalability*) qui traduit le comportement d'un algorithme parallèle lorsque le nombre de processeurs augmente. Là encore, deux manières de voir les choses : on augmente le nombre de processeurs en gardant le même problème ou bien on augmente la taille du problème proportionnellement au nombre de processeurs. Cette deuxième approche permet de conserver une charge constante par processeur. Le principal argument en faveur de cette méthode est que l'on utilise rarement une grosse machine parallèle pour résoudre un petit problème et qu'il est souvent impossible de résoudre un gros problème sur un petit nombre de processeurs en raison du manque de mémoire.

Le parallélisme peut donc encore apporter beaucoup : il doit permettre de résoudre avec le même temps de calcul des problèmes plus importants qu'il était impossible de résoudre avec des machines classiques.

1.7 Conclusion

Seules les machines parallèles permettent actuellement une augmentation sans limites des performances, pourquoi sont-elles si peu utilisées dans l'industrie ? Ce n'est pas à cause de leur prix comme on l'a déjà vu. Peut-être est-ce parce qu'elles manquent de systèmes d'exploitation et d'environnement de programmation conviviaux. Il est sûr que le fait que l'on ne sache actuellement pas cacher le parallélisme à l'utilisateur à cause du manque d'efficacité des techniques de parallélisation automatique est un frein important à leur plus large diffusion.

Les difficultés qui se posent au programmeur qui a choisi de s'investir dans l'utilisation d'une machine parallèle et d'en tirer un bénéfice raisonnable sont :

- Les communications coûtent cher, il faut donc minimiser leur coût, soit en diminuant leur nombre et leur volume, soit en les recouvrant par du calcul. Cela demande un important travail tant au niveau de l'analyse numérique que de l'algorithmique pour concevoir de nouvelles méthodes adaptées à ce type de contraintes.

- La qualité d'un programme parallèle se mesure en partie par son temps d'exécution mais pas uniquement. La granularité des machines parallèles n'étant pas encore stabilisée, il faut que les programmes puissent s'adapter : il faut prendre en compte l'extensibilité. Pour cela il faut connaître, outre la taille du problème, le nombre de processeurs utilisés et ce nombre doit intervenir dans le code. On peut, pour exprimer cela, utiliser des poly-algorithmes.

- Il faut également tenir compte de la portabilité lorsque l'on choisit d'utiliser une machine parallèle : la plupart des grands codes numériques sont écrits en Fortran (ou en C) en partie pour des raisons de portabilité mais aussi parce qu'il n'existe actuellement aucun langage parallèle qui offre autant de souplesse dans ce domaine.

- Un autre grand problème de ces machines, que nous ne prendrons que très peu en compte dans ce travail, est le traitement des entrées-sorties. Actuellement deux tendances existent : soit on éparpille les disques sur tous les processeurs mais il faut amener les codes et les données sur ces disques et récupérer les résultats, soit on utilise quelques disques en des points particuliers et les données transitent sur le réseau de la machine pendant l'exécution, ce qui prend du temps et charge le réseau plus qu'il ne devrait l'être.

Dans la suite de cette thèse, nous utiliserons des machines parallèles à mémoire distribuée que l'on programmera par échange de messages explicites.

Chapitre 2

Programmation par échange de messages

Le but de ce chapitre est de présenter de manière plus approfondie la programmation par échange de messages ainsi que les deux bibliothèques que nous utiliserons dans la suite de cette thèse: PVM et MPI.

2.1 Introduction

Le modèle le plus utilisé pour la programmation des machines parallèles à mémoire distribuée est la programmation par échange de messages. Les principales raisons de ce succès sont la simplicité de ce modèle et sa relative facilité d'utilisation. Il est simple dans le sens où il est très proche de ce qui se passe réellement dans les machines parallèles à mémoire distribuée : les processus ne disposent d'aucune ressource partagée et doivent communiquer en échangeant des données sur un réseau. Il est simple à utiliser car la partie du code consacrée aux calculs est écrite dans un langage standard (Fortran 77 ou C en général) qui placent le programmeur dans un contexte familier. On écrit pour chaque processeur de la machine cible un programme qui comprend le code de calcul à effectuer plus les opérations d'échange de données et de synchronisation avec les autres processeurs. On peut obtenir une bonne efficacité car les compilateurs utilisés sont classiques (donc efficaces) et car on peut contrôler de manière fine le déroulement de l'exécution (attente des messages par exemple). Ce modèle restera vraisemblablement le plus utilisé tant que des progrès significatifs n'auront pas été accomplis dans les modes d'application des autres modèles de programmation parallèle. Ce type de modèle est utilisé par de nombreuses bibliothèques portables (PVM [27, 59], Parmacs [11], Express [24], P4 [9]) ou spécifiques à une machine (EUI sur SP1 [5], NX sur Intel [53]), McBryan donne une bonne vue d'ensemble de ces systèmes dans [47].

Dans ce chapitre nous présentons dans une première partie les bibliothèques d'échange de messages en général puis de manière plus approfondie PVM et MPI qui sont les standards actuels et peut-être futurs (voir [63]) de ce type de programmation

et que nous utiliserons pour nos applications.

2.2 Caractéristiques générales

2.2.1 Communications point à point

On appelle communication point à point l'envoi d'un message d'une tâche donnée à une autre. C'est le mode de communication le plus simple à partir duquel on pourra construire d'autres modes de communications plus complexes.

Pour envoyer un tel message, on a besoin de connaître :

- la tâche émettrice,
- la tâche réceptrice,
- les données à envoyer.

Une telle communication est donc composée de deux parties : l'émission du message que l'on peut écrire schématiquement

```
envoi(destination, contenu)
```

et la réception de ce message :

```
reception(source, contenu).
```

Les entités `destination` et `source` peuvent être un numéro de processeur si l'on travaille avec une seule tâche par processeur ou bien un numéro de processeur plus un identificateur de tâche sur le processeur dans le cas de systèmes multi-tâches (cet aspect peut être masqué si le système attribue à chaque tâche un numéro unique sur l'ensemble des processeurs). Les données à envoyer seront stockées dans un tampon, cela permet au processus de calcul de continuer son exécution sans être perturbé par la communication.

Un problème peut survenir si la même tâche envoie plusieurs messages à une même autre tâche. Il se peut que selon le résultat d'un test à l'intérieur de la tâche émettrice ou selon l'encombrement du réseau au moment des émissions, l'ordre d'arrivée des messages ne puisse être connu à l'avance. On ajoute donc souvent un paramètre supplémentaire qui est un identificateur du message et qui permet de gérer les échanges entre deux processeurs donnés.

2.2.2 Communications globales ou structurées

Nous avons vu au paragraphe 1.3.2 qu'il existe d'autres schémas de communication très utilisés comme la diffusion ou le regroupement. Les bibliothèques de communication modernes offrent aux utilisateurs des fonctions spécifiques pour ce

type de communications. Cela offre deux avantages : la portabilité et l'efficacité. On gagne en portabilité car il n'est plus nécessaire d'écrire un algorithme de diffusion adapté à une machine particulière et en efficacité car les constructeurs peuvent optimiser ces fonctions pour leur machine.

Une autre opération globale qui peut être intéressante est la synchronisation : toutes les tâches attendent d'en être au même point pour continuer leur exécution. Cela peut être utile dans le cas où l'on attend un événement extérieur. Ce schéma de communication est un peu particulier car aucune donnée n'est échangée.

Il est aussi utile de définir des sous-ensembles de l'ensemble des tâches que constitue un programme parallèle selon une caractéristique commune, par exemple la participation à un sous-calcul pendant que d'autres tâches effectuent une autre opération. On peut alors étendre toutes les communications globales que l'on vient de voir à ces groupes. Cela permet une plus grande souplesse et un meilleur rendement que la technique de masquage qui était utilisée sur les premières machines parallèles : dans le cas d'une diffusion par exemple, le message était envoyé à toutes les tâches mais seules quelques-unes effectuaient réellement l'opération de réception. Cela n'était pas très optimal en ce qui concerne l'utilisation du réseau.

Un problème peut se poser lorsque l'on utilise la notion de groupe dans un programme parallèle. Il existe deux possibilités pour définir les groupes : une fois pour toutes au début du programme ou bien de manière dynamique, les tâches pouvant alors rejoindre ou quitter un ou des groupes en cours de calcul. Que se passe-t-il si la composition d'un groupe change au cours de l'exécution d'une opération globale sur ce groupe ? Cela peut être laissé au soin du programmeur, sinon une technique classique consiste à synchroniser toutes les tâches avant une opération sur un groupe, ce qui peut être très gaspilleur en temps.

2.2.3 Synchronisation des communications

Une notion nous sera utile dans la suite de cette thèse, il s'agit de la notion de communication **synchrone** ou **asynchrone**. Le processus émetteur peut avoir deux types de comportements lors de l'exécution d'une phase d'émission (voir également la figure 2.1) :

- soit il doit attendre que le récepteur soit prêt à recevoir ses données pour pouvoir effectuer l'envoi, on parle alors d'envoi synchrone,

- soit il envoie ses données sur le réseau et continue son exécution, on parle alors d'envoi asynchrone. L'avantage de cette solution est un gain de temps évident, c'est la méthode utilisée sur la plupart des systèmes récents.

Dans le cas où l'émission est asynchrone, le système doit gérer tous les messages qui peuvent être envoyés et les stocker jusqu'à leur réception par leur destinataire. Il

faut aussi gérer les cas où les capacités de stockage intermédiaire sont insuffisantes et les cas d'erreur lors des transmissions : c'est le problème du contrôle de flux.

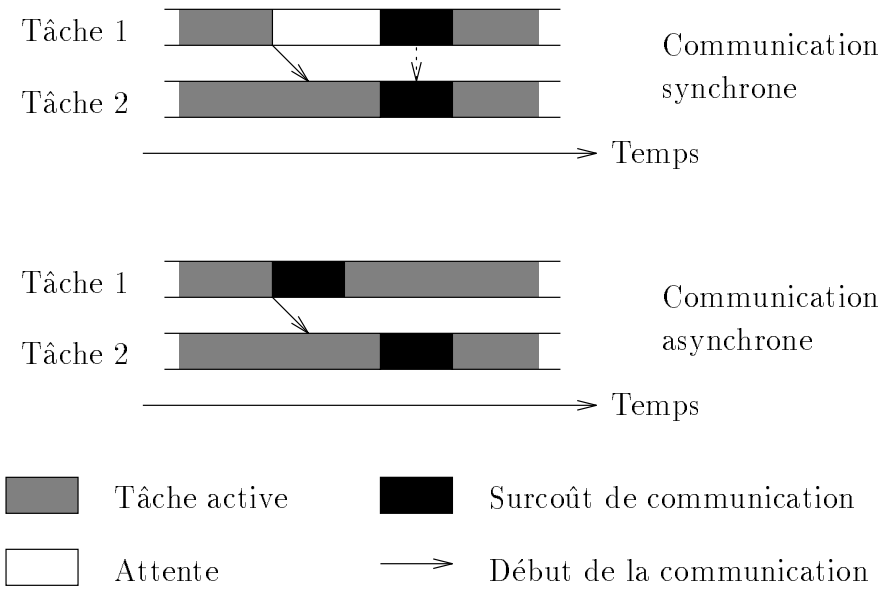


FIG. 2.1 - *Communications synchrones et asynchrones.*

Le processus récepteur peut également avoir un comportement bloquant ou non-bloquant. Son comportement dans le cas d'une réception bloquante est simple, le processus s'arrête jusqu'à l'arrivée des données voulues puis continue son exécution. Dans le cas d'une réception non-bloquante, le processus va tester si un message donné est arrivé (ou si l'émetteur est prêt dans le cas d'une communication synchrone). Si c'est le cas, le message est lu, sinon le processus poursuit son exécution en exécutant un calcul qui ne nécessite pas les données en question par exemple. Lorsque plus aucune activité n'est possible sans les données attendues, le processus effectue une réception bloquante. Les réceptions non-bloquantes permettent donc une plus grande souplesse dans l'exécution des tâches. La plupart des systèmes modernes proposent ce type de réception, en plus des réceptions bloquantes qui sont toujours nécessaires.

2.3 La bibliothèque PVM

2.3.1 Introduction

PVM (Parallel Virtual Machine) est une bibliothèque d'échange de messages développée par une équipe de chercheurs de l'Université du Tennessee et du Oak Ridge National Laboratory. Leur but était à l'origine de pouvoir programmer un réseau de stations de travail (même hétérogène) comme une machine parallèle virtuelle. Les

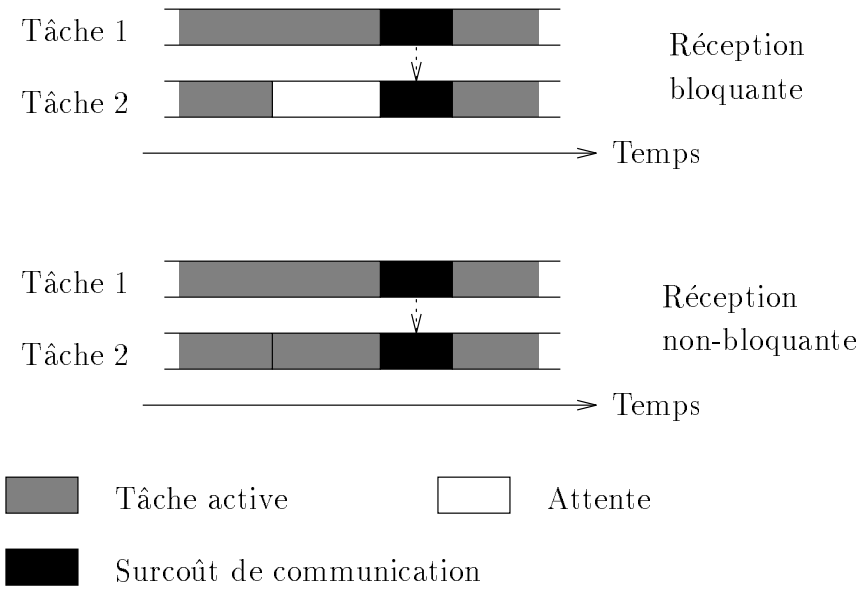


FIG. 2.2 - *Réceptions bloquantes et non-bloquantes.*

intérêts de ce type d'outils sont les suivants :

- Beaucoup d'entreprises disposent de stations de travail qui ne sont pas utilisées au maximum de leur capacité. PVM permet dans ce cas d'utiliser toute la puissance de calcul du réseau pour une grosse application sans investissement supplémentaire. Nous pouvons ainsi atteindre la même puissance de calcul qu'avec une très grosse machine.

- L'avantage des stations de travail par rapport aux machines parallèles est qu'elles disposent d'outils de développement beaucoup plus confortables. Ceci permet notamment un débogage aisé.

Devant le succès de cette bibliothèque, de nombreux constructeurs ont porté PVM sur leur machine de manière optimisée. C'est en particulier le cas d'IBM et de Cray. PVM s'est donc imposé petit à petit comme le standard de la programmation par échange de messages.

Nous avons choisi d'utiliser PVM car au début de ce travail de thèse, nous disposions d'un important réseau de stations de travail, notamment grâce à la collaboration entre le LMC, le CICG[1] et l'Observatoire de Grenoble[2]. De plus, PVM commençait à prendre de l'importance à cette époque, ce qui nous assurait une bonne portabilité de notre travail.

1. Centre Inter-universitaire de Calcul de Grenoble

2. Je tiens à remercier ici Françoise Roch, Pierre Valiron et Laurent Desbat de l'Observatoire de Grenoble ainsi que toute l'équipe du CICG qui ont permis cette collaboration.

2.3.2 L'environnement PVM

PVM est composé de plusieurs parties :

la bibliothèque proprement dite, accessible à l'utilisateur à partir des langages C, C++ et Fortran. La version adaptée au Fortran est une sur-couche d'appel aux routines de PVM qui sont écrites en C.

la console de PVM est une interface qui permet à l'utilisateur d'initialiser sa machine virtuelle et de lancer ses tâches. La console permet une utilisation interactive de PVM.

les démons qui assurent le fonctionnement du système.

Dans la suite de cette partie, nous allons passer en revue les principales fonctionnalités offertes par PVM. Tous les exemples et noms de fonctions de PVM sont donnés pour l'appel en Fortran.

2.3.3 Implémentation

PVM est implémenté en deux parties distinctes. La première partie est un démon (`pvmd3`) qui sera exécuté sur tous les processeurs de la machine virtuelle. Il est écrit de telle manière que tout utilisateur puisse l'installer sur une machine sur laquelle il a un compte. Pour utiliser PVM, l'utilisateur doit configurer sa machine virtuelle au moyen d'un fichier qui contient la liste des hôtes, il lance un démon qui prend ce fichier comme paramètre d'entrée et qui lance les démons sur les autres hôtes. Ces démons coopèrent alors pour émuler une machine parallèle. L'application PVM peut alors être lancée par une commande shell sur n'importe lequel des hôtes.

La deuxième partie de PVM est la bibliothèque (`libpvm3.a`) qui contient toutes les routines d'échange de messages, de lancement et de coordination des tâches et de gestion de la machine virtuelle. L'application doit être *linkée* à cette bibliothèque au moment de la compilation.

Le système PVM a été compilé et testé sur un grand nombre d'architectures (voir [59], page 538, tableau 1), depuis les PC 386 jusqu'au Cray C90 en passant par les machines massivement parallèles. Plusieurs constructeurs proposent également une version spécifique pour leur machine parallèle (IBM, Cray, Convex, Intel, SGI et DEC).

2.3.4 Identification des processus

La première procédure qui doit être appelée dans un programme pour pouvoir utiliser PVM est `pvmfmytid`. Cette procédure inclut la tâche appelante dans le programme parallèle et lui affecte un numéro d'identification. Ce numéro identifie entièrement la tâche dans la machine virtuelle, c'est lui qui sera utilisé pour faire référence à la tâche dans le reste du code.

2.3.5 Gestion de la machine virtuelle

La première étape lorsque l'on veut gérer la machine virtuelle est d'obtenir des informations sur celle-ci. Cela est possible grâce aux appels à `pvmfmstat` et `pvmfconfig`. La première routine fournit l'état d'un processeur de la machine virtuelle et permet de détecter une défaillance éventuelle de ce processeur, la deuxième permet de connaître le nom de tous les processeurs composant la machine virtuelle, leur architecture, leur vitesse relative et le numéro de processus du démon PVM du processeur en question.

On dispose ensuite des instructions `pvmfaddhosts` et `pvmfdelhosts` pour ajouter ou enlever dynamiquement des processeurs à la machine virtuelle. L'ensemble de ces fonctions permet de gérer au mieux l'ensemble des processeurs dont on dispose et de concevoir une application tolérante aux pannes. Cela est important car PVM est conçu pour fonctionner sur des réseaux de stations que l'on ne contrôle pas forcément en totalité.

2.3.6 Gestion des processus

On appelle gestion des processus l'ensemble des opérations qui consistent à créer ou à terminer une tâche et à la placer sur un processeur de la machine virtuelle. Ces fonctions sont accessibles aussi bien de manière interactive par la console PVM que par programme par appel à des fonctions de la bibliothèque. Les principales routines disponibles pour agir sur les processus sont :

`pvmfspawn` pour lancer l'exécution d'une tâche sur la machine virtuelle. On peut préciser le processeur sur lequel la tâche doit être exécutée, le type de processeur dans le cas d'une machine virtuelle hétérogène ou bien laisser le système placer la tâche librement.

`pvmfexit` pour terminer l'exécution de la tâche parallèle courante vis-à-vis du système PVM. En général cette instruction précède le `STOP` qui marque la fin du programme de la tâche.

`pvmfkill` pour tuer une tâche à distance.

`pvmfparent` pour connaître le numéro du processus qui a lancé l'exécution du processus courant. Cela peut être utile pour connaître le numéro du maître dans une application de type maître-esclave.

Il existe également deux instructions permettant de recueillir des informations sur l'état des processus s'exécutant sur la machine. Ces instructions (`pvmfpstat` et `pvmftasks`) sont très semblables aux instructions `pvmfmstat` et `pvmfconfig`, elles donnent accès aux numéros de tâches en cours d'exécution, au nom du fichier exécutable auquel elles correspondent, au numéro de leur tâche parent et à leur état qui indique si la tâche est active ou non.

Cet ensemble d'instructions permet de gérer au mieux les tâches à effectuer. On peut grâce à elles contrôler le placement des tâches pour optimiser les temps de communication. On peut aussi contrôler dynamiquement la charge du système et réguler cette charge.

2.3.7 Les communications

Pour pouvoir échanger des messages entre les processus par PVM, il faut tout d'abord construire ces messages. Pour cela on utilise des zones de mémoire tampon que l'on appellera *buffers*. Pour le programmeur, cela permet de construire plus facilement les messages en accumulant les données à envoyer dans ces zones de mémoire, on appelle cette opération l'**empaquetage** du message. Au niveau de PVM, cela permet de manipuler facilement les messages pour les coder si nécessaire ou bien pour les découper en paquets de taille optimale en fonction du mécanisme de communication utilisé au niveau physique ou logique. Le mécanisme est le même à la réception, le message reçu est stocké dans un buffer puis il est **dépaqueté** par l'utilisateur.

L'envoi d'un message se fait donc par trois appels successifs à PVM de la manière suivante :

```
CALL pvmfinitsend(codage, info)
CALL pvmfpack(type, X, n, saut, info)
CALL pvmfsend(destinataire, etiquette, info)
```

La routine `pvmfinitsend` permet de créer un buffer pour composer le message. Le paramètre `codage` peut prendre plusieurs valeurs pour spécifier si l'on veut utiliser le codage XDR (en cas de communication entre deux processeurs d'architectures différentes), si l'on ne veut aucun codage ou si l'on veut empaqueter seulement l'adresse des données à envoyer. Dans ce dernier cas, il faudra veiller à ne pas modifier les valeurs en question tant que l'envoi du message n'a pas effectivement eu lieu. Le paramètre `info` est un entier qui retourne 0 si l'exécution s'est déroulée correctement, un code d'erreur sinon.

Les paramètres de la routine d'empaquetage `pvmfpack` sont :

`type` est un entier qui indique le type des données à empaqueter. En Fortran, ce type peut être `STRING`, `BYTE`, `INTEGER*2`, `INTEGER*4`, `REAL*4`, `COMPLEX*8`, `REAL*8` ou `COMPLEX*16`.

`X` est la variable qui contient les données à empaqueter.

`n` est un entier qui indique le nombre de données du type défini par le premier paramètre à empaqueter.

`saut` est un entier qui donne l'écart entre deux données successives à empaqueter. Ce paramètre est utile pour empaqueter une ligne d'un tableau par exemple, il aura alors pour valeur la hauteur des colonnes de ce tableau (voir figure 2.3). Il vaudra 1 si on veut empaqueter des données contiguës en mémoire.

`info` est un code d'erreur retourné par la routine.

On peut intercaler plusieurs appels à `pvmfpack` entre `pvmfinitsend` et `pvmfsend`, les données seront alors stockées dans le buffer dans l'ordre de leur empaquetage. Il faudra prendre soin de les dépaqueter dans le même ordre à la réception du message.

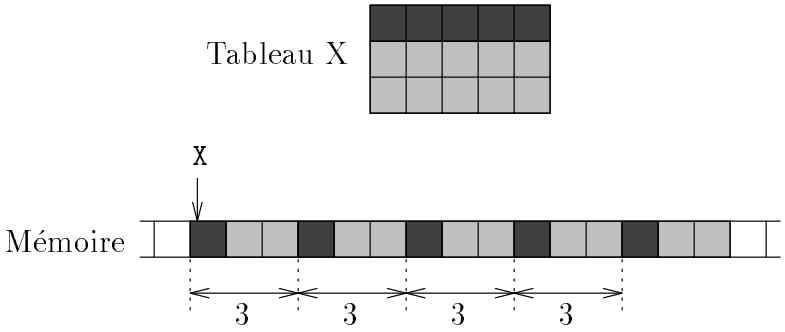


FIG. 2.3 - *Mécanisme d'empaquetage pour* `n=5` *et* `saut=3`.

L'usage de `pvmfsend` est assez simple, ces paramètres sont respectivement le numéro du processus destinataire du message, une étiquette définie par l'utilisateur et qui permet d'identifier le message et un entier qui contient un code de retour.

La réception associée à cet exemple sera :

```
CALL pvmfrecv(source, etiquette, info)
CALL pvmfunpack(type, X, n, saut, info)
```

Les paramètres de `pvmfunpack` sont les mêmes que pour l'empaquetage, ceux de `pvmfrecv` sont les mêmes que ceux de `pvmfsend` à la différence près qu'ici c'est la source du message que l'on doit indiquer. Il existe également une routine `pvmfnrecv` qui a les mêmes paramètres que `pvmfrecv`. La différence entre ces deux routines est que `pvmfrecv` est bloquante alors que `pvmfrecv` est non-bloquante.

En ce qui concerne le contrôle de flux, PVM n'impose aucune limitation sur le nombre de messages ou la taille des buffers. L'utilisateur doit prendre garde à ne pas dépasser les ressources physiques de la machine qu'il utilise.

2.3.8 Autres fonctionnalités

Pour un peu plus de souplesse dans l'utilisation, on peut utiliser la valeur -1 à la place de la source et/ou de l'étiquette dans une réception, cela signifiera que n'im-

porte quelle valeur sera acceptée. Si plusieurs messages répondent à la demande, le premier arrivé est retourné.

PVM offre également deux instructions pour effectuer des diffusions, ce sont `pvmfmcast` et `pvmfbcast`. La première effectue une diffusion sur tous les processus dont les numéros d'identification sont passés en paramètre, la deuxième effectue une diffusion sur un groupe de tâches. Il existe aussi une série d'instructions pour gérer les groupes que nous ne détaillerons pas ici. Notons que l'on peut également définir des barrières de synchronisation entre les processus d'un même groupe.

Une autre possibilité intéressante est de choisir le routage que l'on désire utiliser. Par défaut, le fonctionnement de PVM est le suivant : considérons par exemple une tâche A qui envoie un message à une tâche B. La routine d'envoi de la tâche A envoie son message au démon de sa machine, par l'intermédiaire de sockets UNIX, avec en en-tête la référence de la tâche B. C'est le démon qui recherche la machine sur laquelle se trouve cette tâche et qui envoie le message au démon correspondant par l'intermédiaire du réseau de communication (Ethernet, Token Ring, FDDI, ...) et du protocole de communication UDP/TCP. Le démon de la machine où se trouve la tâche B reçoit le message et le stocke jusqu'à ce que la tâche B le lise par un appel à la routine de réception (voir figure 2.4). Lorsque les deux tâches se trouvent sur la même machine, le message passe par le démon qui le stocke.

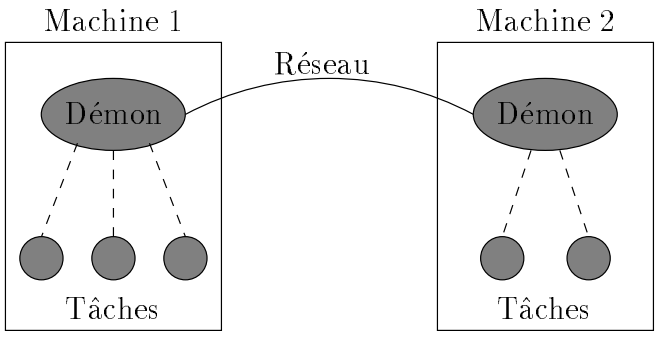


FIG. 2.4 - *Mécanisme de communication de PVM.*

Il est possible (à l'aide d'un appel à `pvmfsetopt`) de demander une communication directe d'une tâche à une autre. Cela est à éviter si la machine virtuelle est hétérogène car les communications deviennent alors moins fiables. Cependant, le gain obtenu par cette méthode peut atteindre 50% du temps de communication.

2.3.9 Conclusions

Tous les constructeurs de machines parallèles à mémoire distribuée fournissent maintenant des versions de PVM optimisées pour leur machine. On peut donc espérer avoir un outil portable et efficace. Cela est cependant à nuancer. Par exemple, IBM

propose une version appelée PVMe optimisée pour ses machines SP1 et SP2 mais cette implémentation ne permet de placer qu'une seule tâche par processeur. Cray propose également une version de PVM optimisée pour son T3D mais elle fonctionne uniquement pour des programmes dits SPMD, c'est-à-dire que tous les processeurs exécutent le même code. Ces versions comportent donc des restrictions importantes par rapport à la version standard de PVM.

2.4 La bibliothèque MPI

2.4.1 Introduction

La bibliothèque d'échange de messages MPI (Message Passing Interface) est née d'une initiative de chercheurs entourés de la plupart des constructeurs de machines parallèles et de quelques industriels à la fin de l'année 1992. Ce travail est parti du constat que le modèle de programmation par échange de messages est suffisamment mûr pour être standardisé. Leur objectif était de standardiser une interface d'échange de messages en essayant de synthétiser les meilleurs aspects de toutes les bibliothèques déjà existantes. Le but de cette opération est de fournir aux utilisateurs une interface portable et facile d'utilisation.

La première version de MPI a été achevée en avril 1994 avec la sortie du rapport du MPI Forum [49]. Cette première version comprend notamment des spécifications pour :

- les communications point à point,
- les opérations globales,
- les groupes de tâches,
- des contextes de communication,
- des topologies logiques de processus,
- des interfaces pour Fortran 77 et C.

Il est important de noter que le forum MPI travaille à la définition d'une interface pour l'échange de messages mais ne propose aucune solution pour leur mise en œuvre. Ce travail est laissé aux constructeurs de machines et à certaines équipes de recherche qui sont intéressées par ces techniques. Il existe actuellement quelques versions implémentées de MPI mais toutes ne sont pas complètes. Nous fournissons ici une vue des fonctionnalités les plus importantes de MPI.

2.4.2 Définitions, généralités

MPI ne comprend aucune fonction de gestion des tâches contrairement à PVM, c'est uniquement une spécification d'interface d'échanges de messages. Le nombre de tâches est fixé par ailleurs au lancement du programme et il est impossible de modifier dynamiquement ce nombre. Il existe des interfaces Fortran et C, les fonctions données ici sont celles de Fortran.

`MPI_INIT` : débute l'utilisation de MPI dans un programme,

`MPI_FINALIZE` : termine un programme MPI,

`MPI_COMM_SIZE` : détermine le nombre de tâches dans le programme,

`MPI_COMM_RANK` : détermine le numéro d'identification de la tâche appelante,

`MPI_SEND` : envoie un message,

`MPI_RECV` : reçoit un message.

Il existe d'autres fonctions de communication point à point, en particulier des versions asynchrones, une version qui permet d'effectuer l'envoi au moment où le récepteur est prêt, ce qui permet d'économiser un buffer de réception.

2.4.3 Communications globales

Elles peuvent être programmées à partir des envois/réceptions de messages mais pour de meilleures performances, il existe des opérations globales (optimisées à l'implémentation). Les principales sont :

`MPI_BARRIER` : synchronise toutes les tâches,

`MPI_BCAST` : diffuse un message à tous les processeurs,

`MPI_GATHER` : regroupe des données de toutes les tâches sur une seule,

`MPI_SCATTER` : distribue des données d'une tâche à toutes les autres (chaque tâche reçoit des données différentes),

`MPI_REDUCE` : recombine (c'est-à-dire regroupe en effectuant une opération sur les données) sur la tâche appelante, l'opération peut être l'une des opérations de base ou définie par l'utilisateur (`MPI_USER_REDUCE`),

`MPI_ALLREDUCE` : effectue des réductions sur tous les processeurs.

2.4.4 Groupes, contextes et communicateurs

La notion d'étiquette n'est pas suffisante pour garantir le bon déroulement des communications : en effet, plusieurs bibliothèques ou plusieurs appels à la même bibliothèque peuvent utiliser la même étiquette à l'intérieur de la même tâche. Pour cela, toutes les opérations de communications dans MPI ont dans leurs arguments un *communicateur* qui contient le groupe de tâches concerné par la communication et un contexte. Deux codes utilisant des contextes différents ne peuvent pas communiquer entre eux et donc ne peuvent pas se perturber mutuellement. Cette notion permet aussi les communications à l'intérieur d'un groupe de tâches sans interférer avec l'extérieur. Il existe quatre fonctions pour agir sur les communicateurs :

`MPI_COMM_DUP` : crée un nouveau communicateur avec le même groupe et un nouveau contexte,

`MPI_COMM_SPLIT` : crée un nouveau communicateur comprenant seulement un sous-groupe d'un groupe donné,

`MPI_INTERCOMM_CREATE` : crée un inter-communicateur qui relie les tâches de deux groupes,

`MPI_COMM_FREE` : détruit un communicateur.

Par exemple, `MPI_COMM_DUP` permet de créer un nouveau communicateur que l'on peut passer en argument à une bibliothèque et qui évitera le mélange entre des messages créés par deux appels distincts à cette bibliothèque. `MPI_INTERCOMM_CREATE` permet les communications point à point entre les tâches de deux groupes (mais pas les opérations globales).

2.4.5 Types de données dérivés

Afin d'éviter des copies de données, des fonctions permettent de définir des ensembles de données à grouper dans un message. Ces fonctions sont par exemple :

`MPI_TYPE_CONTIGUOUS` : groupe un nombre déterminé de données d'un type simple ou dérivé contiguës en mémoire,

`MPI_TYPE_VECTOR` : groupe un nombre déterminé de données d'un type simple ou dérivé espacées régulièrement (exemple : distribution par blocs d'un vecteur ou d'une matrice),

`MPI_TYPE_INDEXED` : groupe un nombre déterminé de données d'un type simple ou dérivé dont les positions sont précisées dans un tableau,

`MPI_TYPE_COMMIT` : active un type de donnée dérivé avant son utilisation,

`MPI_TYPE_FREE` : libère l'espace occupé par un type dérivé.

On peut également grouper des données de types différents dans un même type dérivé.

2.4.6 Topologies de tâches

Certaines configurations de tâches se retrouvent fréquemment dans les programmes parallèles, on peut citer par exemple la grille dans les méthodes de décomposition de domaine. MPI autorise donc la définition d'une topologie logique des tâches. L'implémentation de MPI sur chaque machine doit ensuite permettre de tirer parti de cette information pour utiliser au mieux les possibilités de la machine cible. Cela permet aussi au programmeur d'effectuer certaines opérations telles que le décalage ou le partitionnement de données, notamment sur les graphes de tâches de type cartésien.

2.4.7 Conclusion

Certains points ont été écartés de cette première version car aucune solution n'émergeait, cela concerne en particulier :

- les opérations explicites de mémoire partagée,
- les opérations qui requièrent des possibilités peu courantes des systèmes d'exploitation, comme par exemple les réceptions commandées par interruption, les appels de procédures à distance et les messages dits actifs.
- des outils de construction de programmes,
- des outils de débogage,
- un support explicite des *threads* ou processus légers,
- le support pour la gestion des tâches,
- les opérations d'entrées/sorties.

Ces différents aspects devraient être inclus dans la prochaine version de MPI à laquelle le MPI-Forum travaille maintenant.

Par rapport aux motivations qui étaient à la base de la création de MPI, on peut déjà constater que cette version est déjà très complète en ce qui concerne les points abordés. S'il est possible de faire de même avec les autres aspects abordés dans MPI-2, cette bibliothèque sera réellement complète. La plupart des constructeurs de machines parallèles étant impliqués dans le MPI-Forum, on peut espérer que MPI sera effectivement portable et efficace sur toutes les machines. Par contre, cette portabilité et cette complétude augmentent fortement la complexité de MPI. On peut par exemple recenser 24 manières d'envoyer un message, il devrait y en avoir plus de 70 dans MPI-2.

2.5 Conclusion

On constate depuis l'apparition de MPI l'existence de deux standards en matière de programmation par échange de messages : MPI qui est le standard officiel et PVM qui est le standard *de facto*. On peut actuellement se demander quelle sera l'évolution de ces deux bibliothèques et laquelle il vaut mieux utiliser si l'on commence à développer une application. L'évolution actuelle semble être un rapprochement de ces deux bibliothèques (voir [28]). Le choix semble encore dicté par la machine cible dont dépend la disponibilité et l'éventuelle efficacité des implémentations de ces bibliothèques.

Chapitre 3

L'assimilation de données

Dans ce chapitre, nous introduisons le problème de l'assimilation des données en météorologie et en océanographie. Nous présentons rapidement les différents algorithmes existants puis nous nous attachons plus en détail à l'assimilation variationnelle et à l'écriture et l'utilisation des modèles adjoints. Nous étendrons la méthodologie de l'écriture des modèles adjoints au cas où le modèle direct est parallèle avec échanges de messages explicites.

3.1 Présentation du problème

3.1.1 Méthodes empiriques

L'existence de données en météorologie date du XV$^{\text{ième}}$ siècle avec l'utilisation du pluviomètre vers 1440 en Corée puis l'invention du baromètre par Torricelli en 1644 et l'apparition du premier thermomètre en 1632. Il a fallu attendre 1820 pour voir la première carte globale du champ de pression mais elle regroupait des données échelonnées sur 50 ans ! L'invention du télégraphe en 1845 permet de réaliser la première carte «à jour» en 1851. Depuis cette époque et jusqu'en 1920, des prévisions sont effectuées manuellement à partir de cartes de pression par des méthodes empiriques (analyse subjective).

Des recherches théoriques ont lieu mais elles ne débouchent sur aucune application à la prévision : les équations de base sont connues dès les années 20 mais elles sont non-linéaires, donc on ne peut pas calculer explicitement leur solutions et on n'a pas les moyens de les résoudre numériquement. L'invention de la radio-sonde dans les années 30 permet des améliorations significatives des méthodes empiriques de prévision. On assiste également à cette époque à une forte demande due à l'apparition de l'aviation. Suite à l'invention de l'ordinateur dans les années 40, Charney, Fjørtoft et Von Neumann réalisent en 1950 la première prévision numérique à partir d'équations simplifiées et obtiennent des résultats de qualité comparable à celle des méthodes empiriques. C'est à partir de ce moment que se pose réellement le problème de l'assimilation numérique des données et que les méthodes d'analyse

subjectives se montrent insuffisantes.

3.1.2 Méthodes numériques

On entre alors dans l'ère de la prévision numérique. C'est essentiellement un problème d'évolution qui requiert la donnée de conditions initiales pour être intégré. Il faut pour cela effectuer une analyse des données avant de traiter le problème de l'évolution proprement dit.

Pourquoi ne dispose-t-on pas des conditions initiales permettant d'intégrer un modèle numérique de l'atmosphère? Connaître les conditions initiales signifie connaître parfaitement l'état de l'atmosphère à un instant donné, c'est-à-dire en tout point et avec une précision infinie. Cela est évidemment impossible pour plusieurs raisons :

- Premièrement car la quantité d'information serait infinie !

- Deuxièmement, les données dont on dispose sont très mal réparties sur la surface du globe : beaucoup de mesures sont disponibles dans les pays dits « développés » mais il en existe très peu sur les océans, les zones désertiques et polaires et plus généralement l'hémisphère sud. De plus, toutes ces mesures ne sont pas simultanées : elles ne définissent pas l'état de l'atmosphère à un instant précis. Même en supposant le modèle discrétisé, on ne dispose d'aucun moyen pour mesurer l'état de l'atmosphère à un instant donné en tous les points de discrétisation.

- L'autre problème qui se pose est que toutes les données mesurées le sont avec une certaine incertitude, le modèle lui-même n'étant pas parfait, ses résultats contiendront aussi une certaine part d'erreur qu'il faut prendre en compte. Pour cela il faut combiner le modèle (par l'intermédiaire d'une solution qu'il fournit) et les données. Le procédé de détermination de la condition initiale ne sera de toute façon qu'une approximation que l'on souhaite la meilleure possible.

L'**analyse** consiste à déduire à partir d'observations sur une période donnée l'état de l'atmosphère le plus proche possible de la réalité à un instant donné. La mesure de la qualité de cet état doit prendre en compte les caractéristiques du modèle pour lequel il servira de condition initiale.

Afin de mieux évaluer la taille du problème, précisons que les données utilisables par un système météorologique sont de plusieurs types, elles proviennent de stations terrestres, de radio-sondes, de navires, d'avions et de satellites. Depuis l'émergence des mesures par satellite, les observations de l'atmosphère ont lieu en continu. Cela représente de l'ordre de 10^5 valeurs mesurées par période de 12 heures.

3.1.3 L'analyse objective

La première analyse objective fut produite par Panofsky en 1949 par une technique d'interpolation polynomiale en deux dimensions. Les premières idées mises en œuvre pour assimiler des données étaient simples et relativement peu coûteuses comme l'interpolation polynomiale, souvent améliorée par l'utilisation de méthodes statistiques. Cette méthode a été utilisée en pratique jusque dans les années 60.

La technique suivante fut de mettre à jour séquentiellement les valeurs des variables du modèle au fur et à mesure de l'arrivée de nouvelles observations (début des années 70) par des méthodes de corrections successives. On commence à faire le lien entre la correction statistique et la dynamique du système : c'est l'apparition du concept d'assimilation de données à quatre dimensions (espace et temps). À ce stade de développement, le procédé d'assimilation était le suivant : d'un côté les prévisions sont effectuées pendant que les observations sont utilisées pour produire l'analyse de l'état du système. À l'issue d'une prévision, on utilise les résultats de l'analyse pour corriger les résultats de la prévision et on utilise ce nouvel état corrigé comme point de départ pour la nouvelle prévision (voir figure 3.1). Les données sont ainsi injectées régulièrement dans le modèle (toutes les 12 heures par exemple). On ne peut pas interpoler directement les données observées car elles comportent des erreurs plus ou moins importantes, mais on peut contourner cet inconvénient en utilisant des méthodes de type moindres carrés.

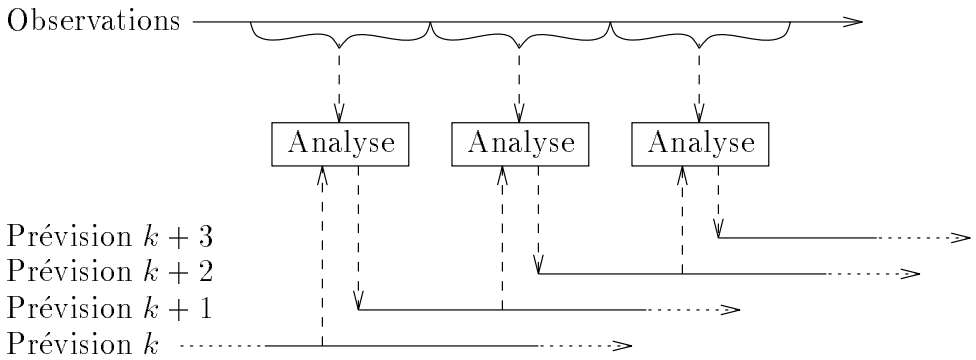


FIG. 3.1 - *Assimilation par corrections successives.*

3.2 Les algorithmes d'assimilation de données

Nous présentons dans cette partie les bases des méthodes d'analyse et d'assimilation de données. Pour plus de détails, on pourra consulter les articles de Ghil et Malanotte-Rizzoli [29] ou de Daley [19]. Nous commencerons par présenter la théorie de l'estimation et la méthode d'interpolation optimale, puis nous introduirons

les deux types de méthodes qui se développent actuellement : le filtre de Kalman et l'assimilation variationnelle.

3.2.1 Méthodes d'estimation

Cas scalaire

Pour mieux comprendre la base de la théorie de l'estimation, considérons un système décrit par une variable d'état scalaire x inconnue. Soit x_o une observation de ce système et x_m un état du système prévu par un modèle. On associe respectivement à x_o et x_m les erreurs ε_o et ε_m, on supposera en outre que ces erreurs sont aléatoires de distribution gaussienne, de moyenne nulle, sans biais, décorrélées entre elles et que leurs variances σ_o et σ_m sont connues. Le but de l'analyse est de fournir x_a, meilleure estimation possible au sens du maximum de vraisemblance de l'état du système.

L'analyse consiste donc à chercher le maximum de la densité de probabilité de x en fonction de x_o, x_m et des variances des erreurs ε_o et ε_m. Le calcul donne :

$$x_a = \frac{\sigma_m^2}{\sigma_o^2 + \sigma_m^2} x_o + \frac{\sigma_o^2}{\sigma_o^2 + \sigma_m^2} x_m$$

dont la variance σ_a vérifie :

$$\frac{1}{\sigma_a^2} = \frac{1}{\sigma_o^2} + \frac{1}{\sigma_m^2}.$$

On remarque que la variance de l'erreur d'analyse est inférieure à la variance de l'erreur d'observation et à la variance de l'erreur du modèle.

Le même résultat peut être obtenu par une approche différente en utilisant un estimateur du minimum de variance de x. Soit un estimateur linéaire sans biais de x défini par $x_e = \alpha_o x_o + \alpha_m x_m$, d'erreur ε_e. Pour que cet estimateur soit sans biais, il faut que $\alpha_o + \alpha_m = 1$. La variance de cet estimateur est donnée par $\sigma_e^2 = \alpha_o^2 \sigma_o^2 + \alpha_m^2 \sigma_m^2$. La minimisation cette variance est équivalente à la minimisation sous la contrainte $\alpha_o + \alpha_m = 1$ de la fonctionnelle :

$$J = \alpha_o^2 \sigma_o^2 + \alpha_m^2 \sigma_m^2 + \lambda(1 - \alpha_o - \alpha_m)$$

où λ est un multiplicateur de Lagrange, ce qui donne :

$$\alpha_o = \frac{\lambda}{2\sigma_o^2} \ , \quad \alpha_m = \frac{\lambda}{2\sigma_m^2} \quad \text{et} \quad \lambda = 2\frac{\sigma_o^2 \sigma_m^2}{\sigma_o^2 + \sigma_m^2}.$$

La variance associée est donnée par :

$$\frac{1}{\sigma^2} = \frac{1}{\sigma_o^2} + \frac{1}{\sigma_m^2},$$

on retrouve donc bien le même résultat que par le calcul du maximum de vraisemblance.

Une troisième forme du même problème consiste à l'écrire sous forme variationnelle. On cherche alors à minimiser la somme des écarts aux observations et à la prévision du modèle de l'état du système. On pondère cette somme par les variances des erreurs respectives des observations et de la prévision du modèle pour tenir compte des incertitudes sur ces valeurs. Cette distance s'écrit donc :

$$J(x) = \frac{1}{2}\left[\sigma_o^2(x - x_o)^2 + \sigma_m^2(x - x_m)^2\right].$$

En annulant la dérivée de cette expression, on retrouve facilement le résultat énoncé ci-dessus, à savoir :

$$x = \frac{\sigma_m^2}{\sigma_o^2 + \sigma_m^2}x_o + \frac{\sigma_o^2}{\sigma_o^2 + \sigma_m^2}x_m.$$

On notera que cette formulation du problème ne fournit pas d'information sur l'incertitude liée à x.

Cas vectoriel

La situation atmosphérique est bien sûr plus compliquée que le cas scalaire. L'atmosphère est décrite par un grand nombre de variables physiques qui sont définies en de nombreux points dans le temps et l'espace. On supposera ici que toutes les observations ont lieu au même instant, on définit alors x_m et x_a comme les valeurs des prévisions du modèle et des résultats de l'analyse sur une grille régulière à cet instant. La taille de ces vecteurs, qui est le nombre de points dans le système, peut atteindre 10^6 en pratique. Les observations x_o ne seront en général pas sur les points de la grille et leur nombre sera différent du nombre de points sur cette grille. Considérons dans un premier temps que les points d'observations coïncident avec les points de la grille. Alors, l'estimateur du maximum de vraisemblance est obtenu en minimisant :

$$I = \frac{1}{2}\left[(x_o - x_a)^t O^{-1}(x_o - x_a) + (x_m - x_a)^t M^{-1}(x_m - x_a)\right]$$

où O et M sont les matrices de covariance d'erreur pour les observations et les prévisions du modèle. La minimisation donne :

$$x_a = O(M + O)^{-1}x_m + M(M + O)^{-1}x_o$$

qui est une généralisation du cas scalaire et qui peut aussi s'écrire :

$$x_a = x_m + M(M + O)^{-1}(x_o - x_m),$$

la matrice de covariance d'erreur vérifiant : $A^{-1} = O^{-1} + M^{-1}$.

Dans le cas général, les observations ne sont pas dans le même espace que les variables du modèle et de l'analyse. Par exemple, pour faire une prévision sur la température, un satellite permet d'observer les rayonnements localisés sur sa trace. On a donc besoin d'un opérateur C pour passer des variables de calcul aux observations. Cet opérateur représente à la fois l'interpolation spatiale et les équations de radiation de l'atmosphère. Il est en général non-linéaire mais peut se linéariser au voisinage du point considéré, nous le supposerons donc linéaire. La fonctionnelle à minimiser devient :

$$I = \frac{1}{2}\left[(x_o - Cx_a)^t O^{-1}(x_o - Cx_a) + (x_m - x_a)^t M^{-1}(x_m - x_a)\right]$$

et le minimum est obtenu pour :

$$x_a = x_m + MC^t(CMC^t + O)^{-1}(x_o - Cx_m), \tag{3.1}$$

la matrice de covariance d'erreur étant donnée par : $A^{-1} = C^t O^{-1} C + M^{-1}$ ou

$$A = M - MC^t(CMC^t + O)^{-1}CM. \tag{3.2}$$

Le terme $x_o - Cx_m$ représente la différence entre les valeurs observées et les valeurs que l'on devrait avoir si l'atmosphère était exactement décrite par l'état du modèle. On appelle ce vecteur le résidu ou vecteur innovation car il contient toute l'information apportée par les observations x_o. On remarquera que la résolution de ces équations nécessite l'inversion de la matrice $CMC^t + O$ dont la dimension est égale au nombre d'observations à l'instant donné, nombre qui est en général très inférieur au nombre de points de discrétisation utilisés.

On peut résoudre ce problème en utilisant sa formulation variationnelle, c'est-à-dire en minimisant la somme pondérée par les matrices de covariance d'erreur des écarts entre l'état analysé du système et les observations d'un côté, la prévision du modèle de l'autre. Cela revient à minimiser la fonctionnelle :

$$J(x) = (x - x_m)^t M^{-1}(x - x_m) + (Cx - x_o)^t O^{-1}(Cx - x_o)$$

La dimension du problème est celle de x, c'est-à-dire le nombre de points de la grille de discrétisation. On montre que le minimum est obtenu pour x_a vérifiant (3.1).

Les formules données ici représentent la forme la plus générale de l'interpolation optimale, qui est la méthode opérationnelle la plus utilisée à l'heure actuelle. Cependant, elle est en général utilisée avec quelques simplifications car d'une part, la dimension des matrices à inverser étant de l'ordre de 10^4 à 10^5, sa résolution exacte serait trop coûteuse et d'autre part la détermination des matrices de covariance d'erreur est difficile. Une approximation souvent employée consiste à utiliser seulement les observations voisines pour faire l'analyse en un point de la grille.

3.2.2 Filtre de Kalman

Les méthodes que nous venons d'exposer permettent de trouver la « meilleure estimation » possible de l'atmosphère à un instant donné compte tenu d'une prévision obtenue à l'aide d'un modèle et d'observations de l'atmosphère à cet instant. Cependant, les observations de l'atmosphère dont on dispose sont réparties dans le temps. On effectue donc plusieurs analyses séquentiellement.

Considérons la situation dans laquelle on a plusieurs observations réparties dans le temps : après avoir réalisé une analyse à partir des observations x_o et d'une prévision x_m donnée par un modèle, on a obtenu une estimation de l'état de l'atmosphère x_a. On intègre alors le modèle à partir de cet état jusqu'à l'instant des observations suivantes x'_o, on obtient x'_m. Si les erreurs associées aux observations x'_o et à la prévision x'_m sont décorrélées alors on peut réutiliser les formules (3.1) et (3.2) pour faire une deuxième analyse. On peut itérer le processus jusqu'à ce que toutes les observations disponibles aient été prises en compte : cette méthode s'appelle le filtre de Kalman [36, 37].

Cet algorithme possède toutes les propriétés d'un bon algorithme d'assimilation : il produit la description la plus précise possible de l'atmosphère compte tenu des sources d'information disponibles ainsi que l'incertitude associée en fonction des incertitudes sur les sources d'information.

La difficulté majeure de la mise en pratique du filtre de Kalman est son coût de calcul. Pour que la méthode soit utilisable, on peut grouper les observations ayant eu lieu dans un intervalle de temps donné au temps milieu de cet intervalle. Cela permet de réduire le nombre de pas de la méthode et donc de limiter le nombre d'analyses à effectuer.

Le principal reproche adressé à l'assimilation séquentielle est qu'une observation donnée influe seulement sur l'évolution future de l'atmosphère mais n'est pas utilisée pour corriger les états passés à cause du caractère en une seule « passe » de l'assimilation séquentielle. On obtient donc une bonne représentation de l'état de l'atmosphère à la fin de la période d'assimilation mais pas sur tout l'intervalle en question.

3.2.3 Assimilation variationnelle

Nous supposerons ici que le modèle décrivant l'atmosphère est parfait, c'est-à-dire que $\varepsilon_m = 0$ ou $M = 0$, l'évolution du système est donc gouvernée par :

$$x_{k+1} = F x_k. \tag{3.3}$$

La formulation variationnelle du problème conduit à considérer la distance définie par :

$$J(x) = \sum_{0 \leq k \leq N} (Cx_k - x_{o,k})^t O^{-1} (Cx_k - x_{o,k}) \tag{3.4}$$

où x est l'ensemble des états du modèle aux instants successifs $k = 0, \ldots, N$, liés entre eux par l'équation d'évolution du modèle (3.3). $J(x)$ est la somme des carrés des écarts entre le modèle et les observations, pondérée par l'inverse de la matrice de covariance d'erreur. Minimiser la distance (3.4) sous la contrainte (3.3) produit pour tout temps k le meilleur estimateur linéaire sans biais de l'état du système compte tenu de toutes les informations disponibles.

En particulier, l'état produit à la fin de la période d'assimilation sera le même que celui produit par le filtre de Kalman (utilisé sans prendre en compte les erreurs du modèle, c'est-à-dire avec $E = 0$). Mais, on remarquera que pour obtenir la formulation variationnelle, il n'est pas nécessaire de faire d'hypothèse de linéarité pour C et F qui peuvent représenter non des matrices mais des opérateurs non-linéaires. Ceci permet de mieux prendre en compte des phénomènes fortement non-linéaires comme les équations de radiation de l'atmosphère en fonction de la température et de l'humidité qui modélisent les observations par satellites.

Dans la suite de ce travail, nous nous intéresserons à cette forme d'assimilation de données.

3.3 Algorithmes de minimisation

Nous avons montré dans les paragraphes précédents que l'assimilation de données variationnelle est en fait un problème de minimisation de grande dimension. Pour le résoudre, nous avons besoin d'algorithmes efficaces de minimisation. Nous nous intéresserons ici à deux types d'algorithmes : la méthode de Newton tronquée et les méthodes de type quasi-Newton.

3.3.1 Méthode de Newton

On sait qu'en tout minimum de la fonctionnelle J, son gradient est nul. Nous allons donc utiliser la méthode de Newton de recherche d'un zéro d'une fonction pour résoudre l'équation

$$\nabla J = 0.$$

On peut écrire la formule de Taylor au premier ordre pour le gradient de la fonction J au voisinage d'un point x :

$$\nabla J(x + p) = \nabla J(x) + H.p + \mathcal{O}(\|p\|^2)$$

où $H = \nabla^2 J(x)$ représente le Hessien de l'application J au point x, on notera $g = \nabla J(x)$ le gradient de J en ce point. Le minimum est atteint pour p vérifiant :

$$\frac{\partial J(x+p)}{\partial p} = 0 \qquad \Longrightarrow \qquad g + Hp = 0.$$

La méthode de Newton est une méthode itérative qui consiste à approcher ∇J au voisinage du point courant par cette formule en négligeant les termes d'ordre supérieur à 1 et à minimiser cette approximation pour trouver l'itéré suivant. Dans $\mathbb{R}^n$, elle s'écrit sous la forme :

$$x_{k+1} = x_k - H_k^{-1} g_k$$

où H_k représente le Hessien et g_k le gradient de J au point x_k. Cela revient donc à chaque étape à former et à résoudre le système :

$$H_k p_k = -g_k, \tag{3.5}$$

p_k sera appelé pas de descente.

On montre que cette méthode converge en une itération si la fonctionnelle J est quadratique, ou si le point de départ est dans un certain voisinage de la solution dans le cas où J est deux fois continuement dérivable.

Il existe de nombreuses variantes de la méthode de Newton qui peuvent s'écrire sous la forme généralisée :

$$x_{k+1} = x_k - A_k^{-1}(x_{k'}) \nabla J(x_k)$$

ce qui donne, suivant les valeurs de $A_k(x_{k'})$ les méthodes suivantes :

- Méthode du gradient à pas fixe pour $A_k(x_{k'}) = \rho^{-1} I$.
- Méthode du gradient à pas variable pour $A_k(x_{k'}) = \rho_k^{-1} I$.
- Méthode du gradient à pas optimal pour $A_k(x_{k'}) = (\rho(x_k))^{-1} I$ où $\rho(x_k)$ est défini par

$$J(x_k - \rho(x_k)\nabla J(x_k)) = min_{\rho \in \mathbb{R}} J(x_k - \rho \nabla J(x_k)).$$

Le principal obstacle à l'utilisation des différentes variantes de cette méthode est leur coût de calcul pour des problèmes de grande taille, car il faut évaluer $A_k^{-1}(x_{k'})$ ou $\nabla^2 J(x_k)$ à chaque itération. Une solution est d'utiliser l'approximation $A_k(x_{k'}) = \nabla^2 J(x_{k'})$ avec $k' = r \times E(k/r)$ c'est-à-dire que l'on garde la même matrice pendant r itérations successives ou même $A_k(x_{k'}) = \nabla^2 J(x_0)$ qui sera valable si le Hessien ne varie pas trop avec k. On peut aussi utiliser les méthodes de Newton tronquées ou les méthodes de quasi-Newton qui sont détaillées dans les paragraphes qui suivent.

3.3.2 Méthode de Newton tronquée

L'idée sur laquelle est basée la méthode de Newton tronquée est que, quand x_k est assez éloigné de la solution, il n'est pas nécessaire de résoudre exactement l'équation de Newton (3.5). On peut raisonnablement penser que la précision du calcul de x_k qui sera le point de départ pour le calcul de x_{k+1} n'aura que très peu d'influence sur la valeur de ce dernier (voir figure 3.2). La dimension de l'équation (3.5) étant grande, on la résout généralement par une méthode itérative (gradient conjugué par exemple). On peut donc tronquer cette résolution à une précision donnée η ou à un nombre d'itérations fixé à l'avance, le but étant de résoudre cette équation avec d'autant plus de précision que l'on se trouve déjà près de la solution (voir [20] pour plus de détails).

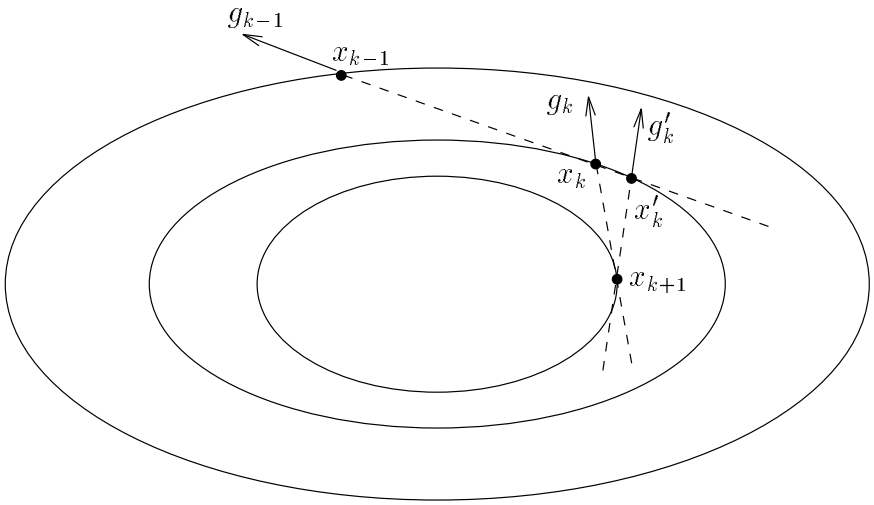


FIG. 3.2 - *Méthode de Newton tronquée.*

Dans la suite de cette thèse, nous utiliserons la méthode du gradient conjugué pour résoudre l'équation (3.5). Dans le cadre qui nous intéresse ici, cette méthode possède l'avantage de ne pas nécessiter la connaissance complète du Hessien, mais uniquement de savoir effectuer le produit du Hessien par un vecteur.

On constate donc que l'on aura besoin des valeurs du gradient de la fonction de coût et du produit du Hessien par un vecteur aux points x_k. Pour obtenir ces valeurs, nous utiliserons les techniques adjointes du premier et second ordre qui seront exposées au paragraphe 3.4.

3.3.3 Méthodes de quasi-Newton

Le principe des méthodes de quasi-Newton est d'utiliser les variations du gradient de la fonction à minimiser entre deux itérés successifs pour obtenir de l'information

sur le Hessien.

Une méthode de quasi-Newton consiste à itérer le processus :

$$\begin{cases} x_{k+1} = x_k - \rho_k H_k^{-1} g_k \\ H_{k+1} = U(H_k, y_k, s_k) \end{cases}$$

où $y_k = g_{k+1} - g_k$ et $s_k = x_{k+1} - x_k$. La deuxième équation peut être remplacée par :

$$H_{k+1}^{-1} = \bar{U}(H_k^{-1}, y_k, s_k)$$

Dans ces équations, ρ_k est un pas positif déterminé par minimisation de J dans la direction $d_k = -H_k^{-1} g_k$. U est un algorithme de mise à jour de H_k^{-1} en fonction de y_k et de s_k. On utilisera aussi la notation $u \otimes v$ définie par :

$$u \otimes v : \mathbb{R}^n \to \mathbb{R}^n : d \to (u \otimes v)d = \langle v, d \rangle u$$

où $\langle ., . \rangle$ est le produit scalaire ordinaire $\langle u, v \rangle = u^t v$ et $u \otimes v = u v^t$.

L'une des méthodes de mise à jour qui donne les meilleurs résultats est la formule BFGS :

$$H_{k+1} = H_k + \frac{y_k \otimes y_k}{\langle y_k, s_k \rangle} - \frac{(H_k s_k) \otimes (H_k s_k)}{\langle H_k s_k, s_k \rangle} \tag{3.6}$$

ou, directement pour l'inverse du Hessien :

$$H_{k+1}^{-1} = H_k^{-1} + \frac{(s_k - H_k^{-1} y_k) \otimes s_k + s_k \otimes (s_k - H_k^{-1} y_k)}{\langle y_k, s_k \rangle} - \frac{\langle s_k - H_k^{-1} y_k, y_k \rangle}{\langle y_k, s_k \rangle^2} s_k \otimes s_k.$$

Ces formules ont la propriété de conserver le caractère défini positif de H_k et H_k^{-1} si et seulement si $\langle y_k, s_k \rangle$ est positif, ce qui est important pour que g_k soit une direction de descente.

En pratique, lorsque la taille du problème est grande, on ne stocke pas H_k ou H_k^{-1} mais seulement un nombre limité de couples (y_k, s_k) et on calcule $H_k^{-1} g_k$ par un algorithme spécialisé. Dans ce cas, la formule inverse est préférable à (3.6) car l'inversion de H_k peut devenir problématique.

Nous utiliserons deux implémentations de cette méthode qui diffèrent par la manière d'initialiser la matrice H_0 : M1QN3 [30] et L-BFGS [44].

3.4 Méthodes adjointes

Nous avons vu que l'assimilation variationnelle de données est un problème de minimisation de l'écart entre les observations et la solution fournie par un modèle.

Les conditions initiales définissant complètement la solution fournie par le modèle, on les choisit comme variables de contrôle et on utilise une méthode itérative pour résoudre ce problème. Il est nécessaire de connaître le gradient de la fonction de coût, on utilise pour cela l'adjoint du modèle considéré.

3.4.1 Principe

Le principe des modèles adjoint est très simple : soit un code qui transforme un vecteur d'entrée u en vecteur de sortie v, on le note :

$$v = G(u). \tag{3.7}$$

Pour une perturbation δu de l'entrée, on obtient une perturbation δv de la sortie telle que

$$\delta v = G'(u).\delta u + \mathcal{O}(\|u\|^2) \tag{3.8}$$

où G' est la matrice des dérivées partielles locales ou matrice Jacobienne. L'équation (3.8) est l'équation linéaire tangente de (3.7). Soit $J(v)$ une fonction scalaire de v. Le gradient de J par rapport à u est donné par :

$$\frac{\partial J}{\partial u_i} = \sum_{j=1}^{n} \frac{\partial v_j}{\partial u_i}\frac{\partial J}{\partial v_j} \quad \text{pour} \quad i = 1, \ldots, q$$

ou, en notation matricielle :

$$\nabla_u J = G'^{*} \nabla_v J. \tag{3.9}$$

La méthode adjointe est basée sur l'utilisation de la formule (3.9). Si l'opérateur G se décompose en une suite d'opérations :

$$G = G_m \circ \ldots \circ G_2 \circ G_1,$$

l'opérateur linéaire tangent s'écrit :

$$G' = G'_m \circ \ldots \circ G'_2 \circ G'_1,$$

et son transposé est :

$$G'^{*} = G'^{*}_1 \circ G'^{*}_2 \circ \ldots \circ G'^{*}_m.$$

Cela montre que pour déterminer le gradient de J par rapport aux entrées u, il « suffit [1] » de parcourir en sens inverse toutes les étapes de G' et d'effectuer à chaque étape la transposée de l'opération considérée appelée aussi son adjoint.

1. Le principe de la méthode est simple mais son application l'est beaucoup moins comme nous le verrons au paragraphe 3.5.

3.4.2 Notations

Les ingrédients de l'assimilation variationnelle sont :

Un modèle: Après la discrétisation en espace (différences finies, éléments finis ou méthodes spectrales), les équations d'évolution de l'atmosphère forment un système d'équations différentielles ordinaires :

$$\frac{dX}{dt} = F(X) \tag{3.10}$$

où X représente le vecteur des variables météorologiques (vent, pression, température, humidité, ...) aux points de discrétisation du modèle. X est la **variable d'état** du système. Avec la condition initiale

$$X(0) = U,$$

l'équation (3.10) a une unique solution sur l'intervalle de temps $[0, T]$. U est la **variable de contrôle** du système, nous noterons $\mathcal{I}$ l'espace des conditions initiales.

Des observations X^{obs} sur l'intervalle $[0, T]$. Pour simplifier les notations, nous considérerons ici que X^{obs} est continu dans le temps sur $[0, T]$. Les observations ne portant pas nécessairement sur les variables météorologiques et afin de pouvoir comparer la variable d'état aux observations, nous introduisons l'opérateur C qui permet de passer de l'espace des variables d'état $\mathcal{X}$ à l'espace des observations $\mathcal{X}^{obs}$. Nous supposerons que C est linéaire.

Une fonction de coût J de $\mathcal{I}$ dans $\mathbb{R}$. Nous la prendrons de la forme :

$$J(U) = \frac{1}{2} \int_0^T \|C.X(t) - X^{obs}(t)\|^2 dt$$

où X est la solution de (3.10) associée à la condition initiale U. J est une mesure de l'écart entre la solution du modèle et les observations.

3.4.3 Contrôle des conditions initiales

Le problème que l'on se pose est de trouver la condition initiale U^* qui minimise J. Une condition nécessaire est d'annuler le gradient de J. Pour calculer ce gradient, on a besoin de la dérivée de Gâteaux de J.

Soit H une perturbation de la condition initiale, la dérivée de Gâteaux $\hat{X}$ de X est la solution de l'équation différentielle :

$$\begin{cases} \dfrac{d\hat{X}}{dt} = \left[\dfrac{\partial F(X)}{\partial X}\right].\hat{X}, \\ \hat{X}(0) = H. \end{cases} \tag{3.11}$$

En météorologie, ce système s'appelle le **système linéaire tangent**.

DÉMONSTRATION : Le système décrivant l'évolution de l'atmosphère est :

$$\begin{cases} \dfrac{dX}{dt} = F(X), \\ X(0) = U. \end{cases}$$

En perturbant la condition initiale dans la direction H, on obtient :

$$\begin{cases} \dfrac{d\tilde{X}}{dt} = F(\tilde{X}), \\ \tilde{X}(0) = U + \alpha H. \end{cases}$$

D'où, en prenant la différence membre à membre :

$$\begin{cases} \dfrac{d(\tilde{X} - X)}{dt} = F(\tilde{X}) - F(X), \\ \tilde{X}(0) - X(0) = \alpha H, \end{cases}$$

soit,

$$\begin{cases} \dfrac{d(\frac{\tilde{X}-X}{\alpha})}{dt} = \dfrac{F(\tilde{X}) - F(X)}{\alpha}, \\ \dfrac{\tilde{X}(0) - X(0)}{\alpha} = H. \end{cases}$$

D'après la définition du Jacobien,

$$F(\tilde{X}) = F(X) + \left[\frac{\partial F}{\partial X}\right](\tilde{X} - X) + O((\tilde{X} - X)^2),$$

donc

$$\lim_{\alpha \to 0} \frac{F(\tilde{X}) - F(X)}{\alpha} = \left[\frac{\partial F}{\partial X}\right] \hat{X}$$

où $\hat{X}$ est la dérivée de Gâteaux de X. On en déduit :

$$\begin{cases} \dfrac{d\hat{X}}{dt} = \left[\dfrac{\partial F(X)}{\partial X}\right].\hat{X}, \\ \hat{X}(0) = H, \end{cases}$$

ce qui achève la démonstration. ∎

La dérivée dans la direction H de la fonction de coût est :

$$\hat{J}(U,H) = \int_0^T (CX - X^{obs}, C\hat{X})dt.$$

Le gradient est calculé en exhibant la dépendance linéaire de $\hat{J}$ par rapport à H. Introduisons la variable adjointe $P \in \mathcal{X}$ dont nous définirons la valeur par la suite. On multiplie (3.11) par P et on intègre sur $[0,T]$. Cela donne :

$$\int_0^T \left(P, \frac{d\hat{X}}{dt}\right) dt = \int_0^T \left(P, \left[\frac{\partial F(X)}{\partial X}\right].\hat{X}\right) dt.$$

On intègre par parties le membre de gauche :

$$\left[\left(P(t), \hat{X}(t)\right)\right]_0^T - \int_0^T \left(\frac{dP(t)}{dt}, \hat{X}(t)\right) dt = \int_0^T \left(\left[\frac{\partial F(X)}{\partial X}\right]^* .P(t), \hat{X}(t)\right) dt$$

d'où

$$\left(P(T), \hat{X}(T)\right) - \left(P(0), \hat{X}(0)\right) = \int_0^T \left(\frac{dP(t)}{dt} + \left[\frac{\partial F(X)}{\partial X}\right]^* .P(t), \hat{X}(t)\right) dt.$$

On choisit P solution du **système adjoint** :

$$\begin{cases} \dfrac{dP}{dt} + \left[\dfrac{\partial F(X)}{\partial X}\right]^* .P = C^*(CX - X^{obs}), \\ \\ P(T) = 0, \end{cases}$$

et l'égalité précédente devient :

$$\begin{aligned} -(P(0),H) &= \int_0^T \left(C^*(C.X - X^{obs}), \hat{X}\right) dt \\ &= \int_0^T \left(C.X - X^{obs}, C.\hat{X}\right) dt \\ -(P(0),H) &= \hat{J}(U,H). \end{aligned}$$

Donc

$$\nabla J(U) = -P(0)$$

et la condition initiale optimale est solution de l'équation

$$\nabla J(U) = -P(0) = 0.$$

La condition initiale optimale est la solution du **système d'optimalité** :

$$\begin{cases} \dfrac{dX}{dt} = F(X), \\ X\,(0) = U, \\ \dfrac{dP}{dt} + \left[\dfrac{\partial F}{\partial X}\right]^{*} .P = C^{*}\left(CX - X^{obs}\right), \\ P(T) = 0, \\ \nabla J(U) = -P(0) = 0. \end{cases}$$

On notera que toute l'information disponible (c'est-à-dire le modèle et les observations) est incluse dans ces équations.

3.4.4 Contrôle des conditions aux limites

Il arrive souvent que la solution d'un modèle météorologique ne dépende pas uniquement des conditions initiales. C'est le cas par exemple des modèles numériques régionaux dont l'intégration nécessite des conditions aux limites. On peut aussi contrôler ces autres entrées du modèle qui peut alors s'écrire :

$$\begin{cases} \dfrac{dX}{dt} = F(X) + B.V, \\ X(0) = U. \end{cases} \tag{3.12}$$

V représente les valeurs de X sur les frontières du domaine d'intégration et dépend de la position et du temps. B est un opérateur linéaire. Pour U et V fixés, le système (3.12) a une solution unique sur $[0, T]$. La fonction de coût est définie par :

$$J(U, V) = \frac{1}{2}\|C.X(U, V) - X^{obs}\|^2.$$

L'ajustement optimal du modèle aux observations est obtenu pour les valeurs initiales U^* et aux limites V^* minimisant $J(U, V)$. Les valeurs U^* et V^* sont caractérisées par l'équation d'Euler :

$$\nabla J(U^*, V^*) = \begin{pmatrix} \dfrac{\partial J}{\partial U} \\ \\ \dfrac{\partial J}{\partial V} \end{pmatrix} = 0.$$

On constate aisément que si l'on choisit P solution du système adjoint :

$$\begin{cases} \dfrac{dP}{dt} + \left[\dfrac{\partial F(X)}{\partial X}\right]^{*} .P = C^{*}(CX - X^{obs}), \\ P(T) = 0, \end{cases}$$

les composantes du gradient sont :

$$\begin{cases} \nabla_U J(U,V) = -P(0), \\ \nabla_V J(U,V) = -B^*.P. \end{cases}$$

DÉMONSTRATION : On multiplie la première équation du système adjoint par $\hat{X}$ et on l'intègre :

$$\int_0^T \left(\frac{dP}{dt}, \hat{X}\right) dt + \int_0^T \left(\left[\frac{\partial F(X)}{\partial X}\right]^* .P, \hat{X}\right) dt$$

$$= \int_0^T \left(C^*(CX - X^{obs}), \hat{X}\right) dt.$$

On intègre par parties le premier terme :

$$\begin{aligned} \int_0^T \left(\frac{dP}{dt}, \hat{X}\right) dt &= \left[\left(P(t), \hat{X}(t)\right)\right]_0^T - \int_0^T \left(P(t), \frac{d\hat{X}(t)}{dt}\right) dt \\ &= -(P(0), \hat{X}(0)) - \int_0^T \left(P(t), \frac{d\hat{X}}{dt}\right) dt. \end{aligned}$$

D'autre part, le système linéaire tangent s'écrit :

$$\frac{d\hat{X}}{dt} = \left[\frac{\partial F(X)}{\partial X}\right] .\hat{X} + B.H_V,$$

donc on peut remplacer le second terme par :

$$\int_0^T \left(\left[\frac{\partial F(X)}{\partial X}\right]^* .P, \hat{X}\right) dt = \int_0^T \left(P, \frac{d\hat{X}}{dt} - B.H_V\right) dt.$$

En reportant le tout dans la première équation, il vient :

$$\begin{aligned} -(P(0), \hat{X}(0)) - \int_0^T (P(t), B.H_V) dt &= \int_0^T \left(C^*(CX - X^{obs}), \hat{X}\right) dt \\ -(P(0), H_U) - (B^*.P, H_V) &= \hat{J}(U,V,H), \end{aligned}$$

d'où le résultat. ■

Les problèmes d'optimisation sous-jacents sont de dimensions différentes. Dans le premier cas, elle est de trois fois le nombre de points de discrétisation; dans le second cas, elle est égale au nombre de points sur la frontière multiplié par le nombre de pas de temps.

On remarque que le système adjoint est le même que dans le paragraphe précédent où l'on s'intéressait aux seules conditions initiales.

3.4.5 Utilisation de l'adjoint au second ordre

Pour calculer la profondeur de descente sur une direction donnée, on a besoin de connaître le Hessien de la fonction de coût, ou plus exactement le produit du Hessien par un vecteur. Pour calculer ce produit, on peut utiliser l'adjoint au second ordre du modèle.

On considère le modèle direct :

$$\begin{cases} \dfrac{dX}{dt} = F(X), \\ \\ X(0) = U, \end{cases}$$

dont l'adjoint s'écrit :

$$\begin{cases} \dfrac{dP}{dt} + \left[\dfrac{\partial F}{\partial X}\right]^* P = C^*(CX - X^{obs}), \\ \\ P(T) = 0. \end{cases}$$

À une perturbation U' de la variable de contrôle correspondent les perturbations $\hat{X}$ de la variable d'état et $\hat{P}$ de la variable adjointe. Elles sont obtenues en dérivant le système direct, ce qui donne le système linéaire tangent :

$$\begin{cases} \dfrac{d\hat{X}}{dt} = \dfrac{\partial F}{\partial X}\hat{X}, \\ \\ \hat{X}(0) = U', \end{cases}$$

et le système adjoint du second ordre :

$$\begin{cases} \dfrac{d\hat{P}}{dt} + \left[\dfrac{\partial F}{\partial X}\right]^* \hat{P} + \left[\dfrac{\partial^2 F}{\partial X^2}\hat{X}\right]^* P = C^* C\hat{X}, \\ \\ \hat{P}(T) = 0. \end{cases}$$

On a déjà vu que :

$$P_U(0) = \nabla_U J \quad \text{et} \quad P_{U+U'}(0) = \nabla_{U+U'} J.$$

Par définition de $\hat{P}$, on a :

$$P_{U+U'}(0) = P_U(0) + \hat{P}(0).$$

La formule de Taylor nous donne :

$$\nabla_{U+U'} J = \nabla_U J + \nabla_U^2 J.U' + \mathcal{O}(\|U'\|^2),$$

d'où on déduit :

$$\hat{P}(0) = \nabla_U^2 J.U'.$$

On obtient donc le produit du Hessien par U' par une intégration rétrograde du système adjoint du second ordre. On peut donc obtenir le Hessien par N intégrations de ce système en prenant successivement les vecteurs de base comme perturbation, mais cela ne sera en général pas nécessaire car, en pratique, le Hessien est utilisé dans une seule direction dans les algorithmes de minimisation. On peut appliquer la même méthode pour obtenir le Hessien par rapport aux conditions aux limites.

Remarque : On peut aussi calculer le produit du Hessien par un vecteur par différence finie entre deux valeurs du gradient. Le coût de calcul serait équivalent mais il n'existe pas de méthode permettant de déterminer la distance optimale entre les deux points de calcul du gradient.

3.4.6 Autres applications des méthodes de contrôle optimal

Identification de paramètres

Certains paramètres des modèles que l'on utilise peuvent être impossibles ou difficiles à mesurer, ils doivent être calculés. C'est le cas, par exemple, de certains coefficients de termes de correction ou des coefficients de diffusion dans l'atmosphère de certains polluants que l'on peut vouloir prendre en compte. Il faut donc estimer au mieux ces paramètres, on peut utiliser pour cela les informations fournies par le modèle adjoint.

On suppose que l'évolution de l'atmosphère est donnée par :

$$\begin{cases} \dfrac{dX}{dt} = F(X,k) + B.V, \\ X(0) = U \end{cases}$$

où k représente l'ensemble des paramètres à évaluer. On suppose que X est parfaitement déterminé lorsque U,V et k sont connus. On cherche les coefficients k minimisant :

$$J(k) = \frac{1}{2}\|C.X(k) - X^{obs}\|^2 = \frac{1}{2}\int_0^T \|C.X(k,t) - X^{obs}(t)\|^2 dt.$$

Pour une perturbation H, le système linéaire tangent s'écrit :

$$\begin{cases} \dfrac{d\hat{X}}{dt} = \left[\dfrac{\partial F}{\partial X}\right]\hat{X} + \left[\dfrac{\partial F}{\partial k}\right] H_k + B.H_V, \\ \hat{X}(0) = H_U. \end{cases}$$

Soit P solution du système adjoint :

$$\begin{cases} \dfrac{dP}{dt} + \left[\dfrac{\partial F}{\partial X}\right]^* P = C.X - X^{obs}, \\ P(T) = 0. \end{cases}$$

En multipliant cette équation par $\hat{X}$ et en intégrant par parties, de la même manière que dans les paragraphes précédents, on obtient :

$$-(P(0), H_U) + \left(\left[\frac{\partial F}{\partial k}\right]^* P, H_k\right) + (B^*P, H_V) = \hat{J}(U, V, k, H).$$

Donc les composantes du gradient sont :

$$\begin{cases} \nabla_U J = -P(0), \\ \nabla_V J = -B^*.P, \\ \nabla_k J = -\left[\dfrac{\partial F}{\partial k}\right]^* P. \end{cases}$$

Le gradient $\nabla_k J$ est utilisé pour optimiser le choix des paramètres k. Le système adjoint reste le même que dans les cas précédents, on applique un opérateur différent aux variables adjointes.

Analyse de sensibilité

Analyser la sensibilité d'un modèle consiste à étudier l'influence d'une perturbation W sur les paramètres k du modèle défini par :

$$\begin{cases} \dfrac{dX}{dt} = F(X, k), \\ X(0) = U. \end{cases}$$

On notera G la solution du modèle perturbé, on a :

$$\hat{G}(k, W) = \left(\frac{\partial G}{\partial X}, \hat{X}\right).$$

Le système linéaire tangent s'écrit :

$$\begin{cases} \dfrac{d\hat{X}}{dt} = \left[\dfrac{\partial F}{\partial X}\right] \hat{X} + \left[\dfrac{\partial F}{\partial k}\right] W, \\ \hat{X}(0) = 0. \end{cases}$$

Soit P solution du système adjoint :

$$\begin{cases} \dfrac{dP}{dt} + \left[\dfrac{\partial F}{\partial X}\right]^* P = \dfrac{\partial G}{\partial X}, \\ P(T) = 0. \end{cases}$$

De la même manière que dans les paragraphes précédents, on multiplie par $\hat{X}$ et on intègre par parties. Cela donne :

$$-\left(\left[\frac{\partial F}{\partial k}\right]^* P, W\right) = \hat{G}(k, W),$$

donc

$$\nabla G = -\left[\frac{\partial F}{\partial k}\right]^* P.$$

Les méthodes adjointes permettent un grand nombre d'applications, le développement du modèle adjoint est donc un investissement utile même en dehors du cadre de l'assimilation de données.

3.5 Détermination pratique du système adjoint

Nous présentons ici une méthode pour dériver l'adjoint et calculer le gradient de la fonction de coût pour un modèle complètement discrétisé. Cette méthode est proche du langage de programmation et sera très utile pour obtenir et implémenter le code correspondant au système adjoint.

3.5.1 Système direct

Nous appliquons ici les techniques décrites précédemment à un modèle discrétisé selon un schéma de différences finies et intégré par la méthode d'Euler. Afin de simplifier un peu l'écriture, nous supposerons que l'on contrôle le système par ses conditions initiales.

Soit X_i l'état de l'atmosphère à l'instant $t_i = t_0 + i\Delta t$ pour $i \in \{0, \ldots, T\}$. Nous avons écrit le modèle représentant l'évolution de l'atmosphère sous la forme :

$$\begin{cases} \dfrac{dX}{dt} = F(X), \\ X(0) = U. \end{cases}$$

La forme discrétisée correspondante est :

$$\left\{\begin{array}{ll} X_0 = U, & \\ X_1 = X_0 & +\Delta t.F(X_0), \\ \vdots & \\ X_t = X_{t-1} & +\Delta t.F(X_{t-1}), \\ \vdots & \\ X_T = X_{T-1} & +\Delta t.F(X_{T-1}). \end{array}\right.$$

Chaque ligne du système ci-dessus est une opération qui peut être non-linéaire et qui sera codée sous forme de boucles `DO`. Nous supposerons dans ce paragraphe que notre modèle est défini par :

$$F(u,v) = \left(\begin{array}{c} u+v \\ u \times v \end{array} \right),$$

ce qui correspond au code :

```
DO i=1,n
  DO j=1,m
    u(i,j,t)=u(i,j,t-1)+dt*(u(i,j,t-1)+v(i,j,t-1))
    v(i,j,t)=v(i,j,t-1)+dt* u(i,j,t-1)*v(i,j,t-1)
  ENDDO
ENDDO
```

L'exemple donné ici représente un modèle bidimensionnel de deux variables `u` et `v`. Il n'a aucune signification physique mais a été choisi dans le but d'expliquer la méthode.

3.5.2 Système linéaire tangent

Nous avons montré dans les paragraphes précédents que le système linéaire tangent est :

$$\left\{\begin{array}{rcl} \frac{d\hat{X}}{dt} & = & \left[\frac{\partial F(X)}{\partial X}\right] \hat{X}, \\ \hat{X}(0) & = & H. \end{array}\right.$$

La forme discrétisée correspondante est :

$$\left\{\begin{array}{ll} \hat{X}_0 = H & , \\ \hat{X}_1 = \hat{X}_0 & +\Delta t. \left[\frac{\partial F(X_0)}{\partial X}\right] \hat{X}_0, \\ \vdots & \\ \hat{X}_t = \hat{X}_{t-1} & +\Delta t. \left[\frac{\partial F(X_{t-1})}{\partial X}\right] \hat{X}_{t-1}, \\ \vdots & \\ \hat{X}_k = \hat{X}_{k-1} & +\Delta t. \left[\frac{\partial F(X_{k-1})}{\partial X}\right] \hat{X}_{k-1}. \end{array}\right.$$

Avec, pour l'exemple précédent :

$$\left[\frac{\partial F(X_{t-1})}{\partial X}\right] = \begin{pmatrix} 1 & 1 \\ v(i,j,t-1) & u(i,j,t-1) \end{pmatrix}.$$

Ici, chaque ligne du système correspond à un produit matrice-vecteur qui sera codé sous forme de boucles `DO` telles que :

```
DO i=1,n
  DO j=1,m
    du(i,j,t) = du(i,j,t-1) + dt *
              ( du(i,j,t-1) + dv(i,j,t-1) )
    dv(i,j,t) = dv(i,j,t-1) + dt *
              ( du(i,j,t-1)*v(i,j,t-1) + u(i,j,t-1)*dv(i,j,t-1) )
  ENDDO
ENDDO
```

Ou bien, avec la convention que l'on garde le nom des variables du système direct pour désigner leurs dérivées et que l'on nomme `z*` la variable `*` du système direct :

```
DO i=1,n
  DO j=1,m
    u(i,j,t) = u(i,j,t-1) + dt *
             ( u(i,j,t-1) + v(i,j,t-1) )
    v(i,j,t) = v(i,j,t-1) + dt *
             ( u(i,j,t-1)*zv(i,j,t-1) + zu(i,j,t-1)*v(i,j,t-1) )
  ENDDO
ENDDO
```

Cette convention est très pratique car elle permet d'écrire le code linéaire tangent à partir du code direct en apportant très peu de modifications. Cela permet de gagner du temps et surtout d'éviter beaucoup d'erreurs de codage.

3.5.3 Système adjoint

Nous avons vu que le système adjoint s'écrit :

$$\begin{cases} \dfrac{dP}{dt} + \left[\dfrac{\partial F(X)}{\partial X}\right]^* .P = C^*(CX - X^{obs}), \\ P(T) = 0. \end{cases}$$

Sous une forme plus proche du code, cela peut s'écrire :

$$\left\{\begin{array}{ll} P_k = C_k & , \\ P_{k-1} = P_k & +\Delta t.\left[\frac{\partial F(X_{k-1})}{\partial X}\right]^* P_k + C_{k-1}, \\ \vdots & \\ P_{t-1} = P_t & +\Delta t.\left[\frac{\partial F(X_{t-1})}{\partial X}\right]^* P_t + C_{t-1}, \\ \vdots & \\ P_0 = P_1 & +\Delta t.\left[\frac{\partial F(X_0)}{\partial X}\right]^* P_1 + C_0, \end{array}\right.$$

avec, pour l'exemple précédent :

$$\left[\frac{\partial F(X_{t-1})}{\partial X}\right]^* = \begin{pmatrix} 1 & v(i,j,t-1) \\ 1 & u(i,j,t-1) \end{pmatrix}.$$

Chaque opération matricielle de la ligne précédente sera codée sous forme de boucles `DO` telles que :

```
DO i=1,n
  DO j=1,m
    p(i,j,t-1) = p(i,j,t)
                 + dt*( p(i,j,t) + v(i,j,t-1)*q(i,j,t) ) + cp(t-1)
    q(i,j,t-1) = q(i,j,t)
                 + dt*( p(i,j,t) + u(i,j,t-1)*q(i,j,t) ) + cq(t-1)
  ENDDO
ENDDO
```

Ou bien, avec la convention que l'on garde le nom des variables du système direct pour désigner leurs adjoints et que l'on nomme `z*` la variable `*` du système direct :

```
DO i=1,n
  DO j=1,m
    u(i,j,t-1) = u(i,j,t)
               + dt*( u(i,j,t) + zv(i,j,t-1)*v(i,j,t) ) + cp(t-1)
    v(i,j,t-1) = v(i,j,t)
               + dt*( u(i,j,t) + zu(i,j,t-1)*v(i,j,t) ) + cq(t-1)
  ENDDO
ENDDO
```

Cette méthode ne permet pas d'obtenir facilement le code adjoint que l'on cherche car il n'y a aucun lien évident entre le code obtenu et le code direct ou le code linéarisé. Nous allons donc voir une autre méthode et montrer sur un exemple que le code obtenu est équivalent au précédent.

3.5.4 Écriture du code adjoint

D'après le calcul du paragraphe 3.4.1, on peut obtenir le code du système adjoint plus simplement. En effet, on a vu dans ce paragraphe que l'adjoint d'une fonction

composée n'est autre que la composition des adjoints des étapes élémentaires prises dans l'ordre inverse. Cette méthode peut donc se résumer en trois étapes :

1. Dériver les instructions.
2. Inverser l'ordre des instructions, en termes mathématiques, cela correspond à la transposition par rapport au temps.
3. Transposer chaque instruction.

Remarque : Inverser l'ordre des instructions correspond à la transposition par rapport au temps.

Inverser l'ordre des instructions

Dans le cas d'une séquence d'instructions arithmétiques, inverser l'ordre est simple. Cette opération peut être plus compliquée dans le cas de boucles ou de tests.

Une boucle peut être écrite sous forme déroulée, c'est-à-dire que l'on peut écrire n fois la séquence d'instructions contenue dans la boucle. On peut alors inverser l'ordre des instructions et on constate que cela revient à inverser l'ordre des instructions dans la boucle et à effectuer la boucle dans le sens inverse. Par exemple, la boucle :

```
DO i=1,3
  Instruction1(i)
  Instruction2(i)
ENDDO
```

devient :

```
  Instruction1(1)
  Instruction2(1)
  Instruction1(2)
  Instruction2(2)
  Instruction1(3)
  Instruction2(3)
```

En inversant les instructions, on obtient :

```
  Instruction2(3)
  Instruction1(3)
  Instruction2(2)
  Instruction1(2)
  Instruction2(1)
  Instruction1(1)
```

Ce qui revient à :

```
DO i=3,1,-1
  Instruction2(i)
  Instruction1(i)
ENDDO
```

Une instruction de test représente une alternative entre deux séquences possibles. Si l'on garde une trace du chemin suivi à l'exécution du code direct, inverser l'ordre des instructions n'est pas une difficulté. Donc pour coder le système adjoint, il faut «se rappeler» du chemin suivi lors de l'exécution du système direct, c'est-à-dire conserver toutes les valeurs ayant servi à faire des choix et refaire les mêmes choix dans le code adjoint. Par exemple, si le code linéarisé s'écrit :

```
IF (a.gt.b) THEN
  Instruction1
  Instruction2
ELSE
  Instruction3
  Instruction4
ENDIF
```

Si à l'exécution on a `a>b`, on exécutera :

```
  Instruction1
  Instruction2
```

Ce qui donnera dans le code adjoint :

```
  Instruction2
  Instruction1
```

et qui revient à :

```
IF (a.gt.b) THEN
  Instruction2
  Instruction1
ELSE
  Instruction4
  Instruction3
ENDIF
```

où `a` et `b` sont **les valeurs de** `a` et `b` obtenues lors de l'exécution du **code direct**.

Transposer une instruction élémentaire

Les instructions de calcul d'un code sont toujours de la forme : une variable reçoit une valeur dépendant de n variables, cette opération est notée par un signe `=` mais c'est en fait une instruction d'affectation dont les arguments sont toutes les variables qui apparaissent dans le terme de droite. La transposée d'une application de $\mathbb{R}^n$ dans $\mathbb{R}$ (on se trouve bien dans ce cas puisque le code a déjà été linéarisé, voir page 74) étant une application de $\mathbb{R}$ dans $\mathbb{R}^n$, la transposée de cette instruction sera composée de n instructions. Par exemple, l'instruction :

```
a = b * c
```

est linéarisée dans une première étape, elle devient (avec la convention que l'on garde le nom des variables du système direct pour désigner leurs dérivées et que l'on nomme `z*` la variable `*` du système direct) :

```
a = b * zc + zb * c
```

L'ordre des instructions est ensuite inversé, puis on transpose cette instruction, ce qui donne :

```
q = q + p * zc
r = r + p * zb
```

où `(p,q,r)` est l'adjoint de `(a,b,c)`. En nommant l'adjoint de chaque variable par son nom dans le système direct, on obtient une deuxième forme équivalente :

```
b = b + a * zc
c = c + a * zb
```

En effet, matriciellement, cela s'écrit simplement. L'instruction linéarisée correspond à l'opération :

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \longleftarrow \begin{pmatrix} 0 & zc & zb \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} b \times zc + c \times zb \\ b \\ c \end{pmatrix}$$

La transposition donne :

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \longleftarrow \begin{pmatrix} 0 & zc & zb \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^* \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ zc & 1 & 0 \\ zb & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 0 \\ b + a \times zc \\ c + a \times zb \end{pmatrix}$$

On peut noter que l'application systématique de cette méthode engendre des lignes de code inutiles qu'il sera intéressant d'éliminer par la suite.

3.5.5 Exemple

Reprenons notre exemple des paragraphes précédents. Notre code linéaire était :

```
DO i=1,n
  DO j=1,m
    u(i,j,t) = u(i,j,t-1) + dt *
             ( u(i,j,t-1) + v(i,j,t-1) )
    v(i,j,t) = v(i,j,t-1) + dt *
             ( u(i,j,t-1)*zv(i,j,t-1) + zu(i,j,t-1)*v(i,j,t-1) )
  ENDDO
ENDDO
```

La méthode que l'on vient de voir nous donne pour l'adjoint :

```
DO i=n,1,-1
  DO j=m,1,-1

c   v(i,j,t) = v(i,j,t-1) + dt *
c            ( u(i,j,t-1)*zv(i,j,t-1) + zu(i,j,t-1)*v(i,j,t-1) )

    v(i,j,t-1) = v(i,j,t-1) + dt*zu(i,j,t-1)*v(i,j,t)
    u(i,j,t-1) = u(i,j,t-1) + dt*zv(i,j,t-1)*u(i,j,t)
    v(i,j,t-1) = v(i,j,t-1) + v(i,j,t)

c   u(i,j,t) = u(i,j,t-1) + dt *
c            ( u(i,j,t-1) + v(i,j,t-1) )

    v(i,j,t-1) = v(i,j,t-1) + dt*u(i,j,t)
    u(i,j,t-1) = u(i,j,t-1) + dt*u(i,j,t)
    u(i,j,t-1) = u(i,j,t-1) + u(i,j,t)

  ENDDO
ENDDO
```

En regroupant les instructions, cela est équivalent à :

```
DO i=n,1,-1
  DO j=m,1,-1
    v(i,j,t-1) = v(i,j,t-1) + v(i,j,t)
                + dt*zu(i,j,t-1)*v(i,j,t) + dt*u(i,j,t)
    u(i,j,t-1) = u(i,j,t-1) + dt*zv(i,j,t-1)*u(i,j,t)
                + u(i,j,t) + dt*u(i,j,t)
  ENDDO
ENDDO
```

En simplifiant les expressions, on obtient :

```
DO i=n,1,-1
  DO j=m,1,-1
    v(i,j,t-1) = v(i,j,t-1) + v(i,j,t)
                 + dt*( u(i,j,t) + zu(i,j,t-1)*v(i,j,t) )
    u(i,j,t-1) = u(i,j,t-1) + u(i,j,t)
                 + dt*( u(i,j,t) + zv(i,j,t-1)*u(i,j,t) )
  ENDDO
ENDDO
```

Il ne faut pas oublier que la fin du code linéaire tangent contient des instructions d'affectation des valeurs du gradient que l'on va retrouver transposées au début du code adjoint et qui vont initialiser `u` et `v` aux valeurs de $C^*(CX - X^{obs})$ qui étaient notées `cp(t-1)` et `cq(t-1)` dans le paragraphe 3.5.3. Les deux codes obtenus sont donc bien équivalents.

3.5.6 Système adjoint du second ordre

Le code du système adjoint du second ordre est obtenu en dérivant le code du système adjoint. On doit pour cela conserver les trajectoires des systèmes direct, tangent et adjoint. On notera `z*` la trajectoire de la variable `*` dans le code direct, `t*` la trajectoire de la variable `*` dans le code tangent et `a*` la trajectoire de la variable `*` dans le code adjoint. Reprenons notre exemple précédent, le code adjoint que nous avions obtenu est :

```
DO i=n,1,-1
  DO j=m,1,-1
    v(i,j,t-1) = v(i,j,t-1) + v(i,j,t)
                 + dt*( u(i,j,t) + zu(i,j,t-1)*v(i,j,t) )
    u(i,j,t-1) = u(i,j,t-1) + u(i,j,t)
                 + dt*( u(i,j,t) + zv(i,j,t-1)*u(i,j,t) )
  ENDDO
ENDDO
```

En dérivant de nouveau ce code, on obtient l'adjoint du second ordre :

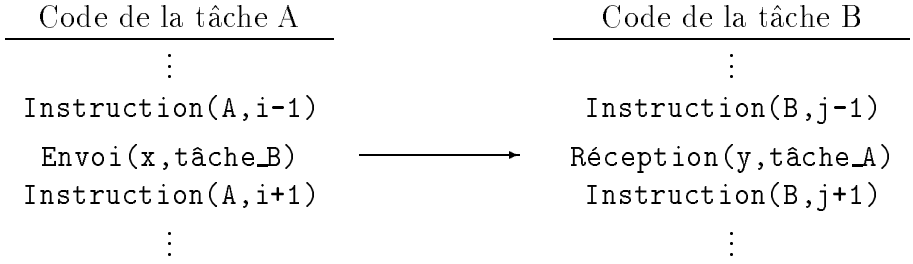
```
DO i=n,1,-1
  DO j=m,1,-1
    v(i,j,t-1) = v(i,j,t-1) + v(i,j,t) + dt*( u(i,j,t)
                 + tu(i,j,t-1)*av(i,j,t) + zu(i,j,t-1)*v(i,j,t) )
    u(i,j,t-1) = u(i,j,t-1) + u(i,j,t) + dt*( u(i,j,t)
                 + tv(i,j,t-1)*au(i,j,t) + zv(i,j,t-1)*u(i,j,t) )
  ENDDO
ENDDO
```

3.5.7 Adjoint d'un code parallèle

Dans le cadre de cette thèse, nous avons été amenés à écrire l'adjoint d'un modèle dont le code direct avait été parallélisé par l'utilisation d'une bibliothèque d'échange de messages. Nous présentons donc dans ce paragraphe les transformations que doivent subir les instructions d'échanges de messages lors de l'écriture du code adjoint.

Adjoint d'un envoi de donnée

La première fonctionnalité spécifique que l'on peut trouver dans un code parallèle est l'envoi d'une donnée d'un processeur à un autre. Nous avons donc un code parallèle qui peut s'écrire :



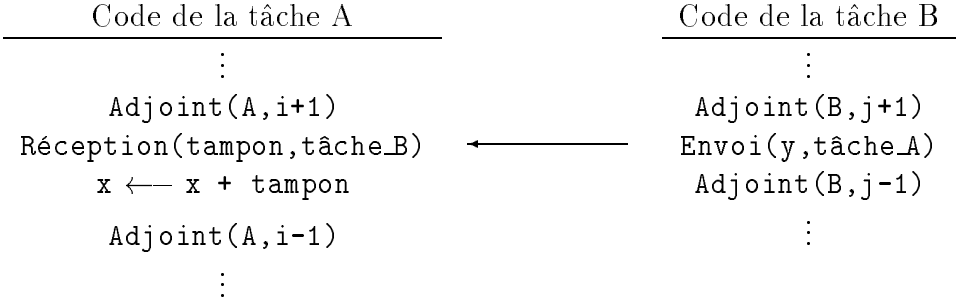
L'envoi de la valeur `x` de la tâche A à la tâche B qui la stocke dans sa variable `y` peut être considéré comme l'affectation de la variable `x(A)` à la variable `y(B)`, schématiquement :

$$\texttt{y} \longleftarrow \texttt{x} \ ,$$

dont nous avons vu que l'opération adjointe est :

$$\texttt{x} \longleftarrow \texttt{x} + \texttt{y}$$

La variable `x` étant stockée et sa valeur utilisée dans la tâche A, il faut d'abord envoyer la valeur de `y` de la tâche B à la tâche A pour pouvoir effectuer cette opération. L'adjoint du code parallèle ci-dessus est donc :



Opération	Code Direct	Code Adjoint
Envoi	`Envoi(x,Proc)`	`Réception(tampon,Proc)` `x ⟵ x + tampon`
Réception	`Réception(x,Proc)`	`Envoi(x,Proc)`
Synchronisation	`Synchro(Liste_Proc)`	`Synchro(Liste_Proc)`

Diffusion	`Diffusion(x,Liste_Proc)`	`Pour Proc ∈ Liste_Proc`
		`  Réception(tampon,Proc)`
		`  x ⟵ x + tampon`

Réduction	`Pour Proc ∈ Liste_Proc`
	`  Réception(tampon,Proc)`
	`  x ⟵ x + tampon`
		`Diffusion(x,Liste_Proc)`

TAB. 3.1 - *Code adjoint des instructions d'échange de messages.*

Généralisation

Les communications globales étant définies à partir des envois et réceptions de messages, on peut de la même manière définir les adjoints des opérations globales de communications. On obtient ainsi le tableau 3.1.

Dériver l'adjoint d'un code parallèle n'est donc théoriquement pas plus difficile que de dériver l'adjoint d'un code séquentiel et peut s'automatiser de la même façon. Nous étudierons dans la partie 5.2 les performances de cette technique.

3.6 Conclusion

Nous avons rapellé dans ce chapitre l'origine de l'assimilation de données jusqu'à l'émergence des deux types de méthodes employées actuellement : le filtre de Kalman et l'assimilation variationnelle. Nous nous intéressons dans le cadre de ce travail à cette deuxième méthode. L'un de ses intérêts est de permettre d'autres applications que l'assimilation de données elle-même telles que l'estimation de paramètres et l'analyse de sensibilité.

Nous avons ensuite montré comment développer un code adjoint, on constate que ce n'est pas un processus très compliqué. Il peut néanmoins devenir long et fastidieux lorsque l'on veut l'appliquer à de gros codes comme les modèles météorologiques ou océanographiques. Les écrire à la main nécessite une validation à chaque étape et conduit souvent à beaucoup d'erreurs de typographie. Il est donc souhaitable que ce processus soit automatisé. Des travaux sur la différentiation automatique sont en cours (voir [56]) mais ne sont pas encore suffisamment avancés pour permettre une utilisation vraiment automatique, une difficulté importante venant du passage des paramètres d'une procédure à une autre.

Chapitre 4

Parallélisation et couplage de modèles

Le but de ce chapitre est de présenter les différentes approches possibles a priori pour paralléliser la résolution du problème d'assimilation de données. Cela peut se faire à plusieurs niveaux : au niveau des modèles météorologiques direct et adjoint, au niveau de l'algorithme d'optimisation ou enfin au niveau du problème lui-même. Cela signifiera alors transformer un problème séquentiel d'optimisation sans contraintes en un ensemble de problèmes d'optimisation relativement indépendants qui pourront être résolus en parallèle. On étudiera plusieurs variantes de ces trois approches très générales et leur utilité dans le cadre du problème d'assimilation de données.

4.1 Parallélisation des modèles

Lorsque l'on étudie la répartition du temps de calcul entre les différentes phases de l'exécution d'un algorithme variationnel d'assimilation de données en météorologie, on se rend compte que l'algorithme d'optimisation proprement dit demande un très faible pourcentage du temps total. En effet, l'évaluation de la fonction de coût et de ses dérivées requièrent l'intégration du modèle météorologique et de son adjoint (ainsi que l'adjoint au second ordre éventuellement). L'algorithme de minimisation en lui-même nécessite en général peu de calculs (un produit scalaire et quelques mises à jour de vecteurs dans le cas du gradient conjugué). On peut donc se contenter de paralléliser ces parties du code.

L'algorithme consistera donc à itérer deux phases successives :

- Une phase de calcul de la fonction de coût et de ses dérivées. Cette phase peut être parallélisée.
- Une phase d'optimisation proprement dite. Cette phase est peu coûteuse en calculs mais nécessite le regroupement des valeurs de la fonction de coût et de son gradient et la diffusion du nouvel itéré pour le calcul des valeurs suivantes.

Le graphe de tâches correspondant à cette méthode est représenté sur la figure 4.1.

Remarque : Selon l'algorithme de minimisation utilisé, on peut diminuer la taille des messages du regroupement et de la diffusion en effectuant en parallèle certaines opérations de mise à jour des vecteurs et des produits scalaires partiels. On peut ainsi arriver à réduire la taille des messages à quelques nombres réels dans les meilleurs cas.

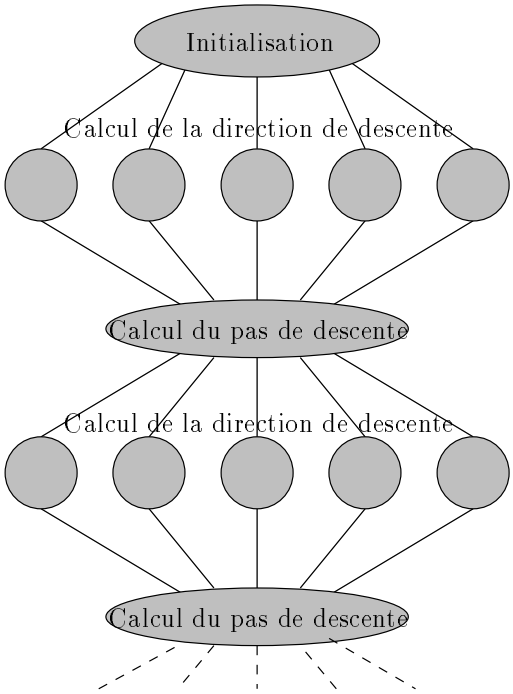


FIG. 4.1 - *Graphe des tâches correspondant à la parallélisation des modèles : la phase de calcul de la direction de descente utilise les modèles direct et adjoints, elle est parallèle; la phase de calcul du pas de descente est centralisée sur un processeur.*

Quel va être le comportement de cet algorithme lorsque le nombre de processeurs augmente ? La première étape doit en principe prendre de moins en moins de temps si les modèles directs et adjoints se parallélisent bien. Le temps de calcul de la deuxième étape ne doit pas varier, par contre son temps de communication va augmenter. Globalement, on aura donc une partie parallèle dont le temps d'exécution diminue et une partie séquentielle dont le temps d'exécution augmente : l'accélération devrait donc se dégrader. Cette dégradation sera sensible dès que les temps de regroupement et diffusion ne seront plus négligeables devant le temps d'intégration des modèles directs et adjoints.

Un avantage de cette méthode est que le modèle direct parallèle est de toutes façons utile pour la prévision. On a vu que le développement de l'adjoint d'un

code parallèle ne demande pas beaucoup plus de travail que le développement du code séquentiel. Cette méthode a donc l'avantage d'être relativement peu coûteuse à mettre en œuvre et l'investissement réalisé est réutilisable.

4.2 Algorithmes parallèles d'optimisation sans contraintes

Pour résoudre les problèmes d'optimisation sans contraintes, il existe de nombreuses méthodes itératives. Après avoir précisé nos hypothèses de travail, nous passerons en revue les principaux algorithmes et nous étudierons leur parallélisation et leur application au problème de l'assimilation de données.

4.2.1 Généralités

De façon générale, le problème de l'assimilation de données peut se ramener au problème d'optimisation sans contraintes :

$$\text{Trouver } u^* \in U \text{ tel que } J(u^*) = \min_{u \in U} J(u).$$

où $J : U = \mathbb{R}^m \to \mathbb{R}$ est une fonctionnelle que l'on supposera différentiable, nous préciserons les hypothèses supplémentaires que doit vérifier J pour l'application de chaque algorithme. Le but de ce paragraphe est de montrer dans quelle mesure cette fonctionnelle peut se décomposer en somme de fonctionnelles relativement indépendantes.

Dans le cadre de l'assimilation de données, la fonctionnelle J peut s'écrire :

$$J(u) = \frac{1}{2} \int_0^T \int_\Omega \|x(u) - x^{obs}\|^2 d\Omega dt$$

où Ω est le domaine géographique sur lequel on fait l'assimilation des données. La variable x est celle qui décrit le système sur le domaine et l'intervalle de temps considérés, elle se nomme **variable d'état**. La variable u est celle par rapport à laquelle on effectue la minimisation, c'est la **variable de contrôle**. On peut déjà remarquer que les méthodes classiques de décomposition ne s'appliquent pas ici à cause des dépendances non locales induites par la relation qui lie x à u.

Dans un premier temps, considérons que la fonction de coût est une fonction de la variable d'état :

$$J(x) = \frac{1}{2} \int_0^T \int_\Omega \|x - x^{obs}\|^2 d\Omega dt,$$

la variable d'état appartenant à un espace de fonctions donné (par exemple les fonctions continues sur Ω). On suppose que Ω se décompose en $n-1$ sous-domaines

Ω_i tels que l'on puisse décomposer l'intégrale :

$$\int_\Omega \|x - x^{obs}\|^2 d\Omega = \sum_{i=1}^{n-1} \int_{\Omega_i} \|x_i - x_i^{obs}\|^2 d\Omega_i,$$

où x_i et x_i^{obs} sont les restrictions respectives des variables d'état x et des observations sur le domaine Ω_i, on a alors

$$J(x) = \sum_{i=1}^{n-1} J_i(x_i) \quad \text{où} \quad J_i(x_i) = \frac{1}{2} \int_0^T \int_{\Omega_i} \|x_i - x_i^{obs}\|^2 d\Omega_i dt.$$

Revenons maintenant à la variable de contrôle u : la variable d'état x est en fait une fonction de $u \in U$, mais sa restriction à un domaine Ω_i ne dépend pas de toutes les composantes du contrôle. En effet, selon la méthode utilisée pour intégrer le modèle qui lie u à x, la matrice de dépendance de x par rapport à u ne sera pas pleine. Cette matrice aura une structure plus ou moins creuse selon le schéma de discrétisation spatial et la méthode d'intégration temporelle utilisés. Pour la suite, on supposera que x_i dépend du contrôle sur Ω_i et sur une partie plus ou moins importante des Ω_j que l'on notera Ω_{ij} pour $i \neq j$ (voir un exemple figure 4.2). On note $\Omega_{n,i}$ la réunion des Ω_{ij} pour $j \neq i$ et on pose :

$$\Omega_n = \bigcup_{i=1}^{n-1} \Omega_{n,i}.$$

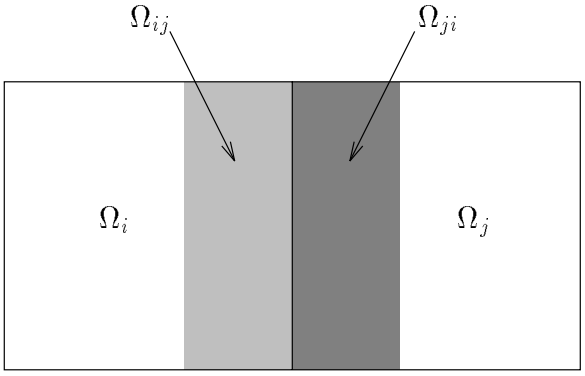


FIG. 4.2 - *Dépendance entre les sous-domaines dans la décomposition de la fonction de coût.*

La figure 4.3 montre un exemple de ce que l'on peut obtenir dans le cas d'un domaine Ω rectangulaire découpé en sous-domaines Ω_i eux aussi rectangulaires. On définit $U_i = \mathbb{R}^{m_i}$ comme l'espace vectoriel des variables de contrôle définies sur $\Omega_i - \Omega_n$ pour $i < n$ et sur Ω_n pour $i = n$. On remarquera que les Ω_i sont des domaines géométriques (des régions géographiques dans le cas de l'assimilation de données en géophysique) et que les U_i sont des espaces vectoriels de dimension m_i

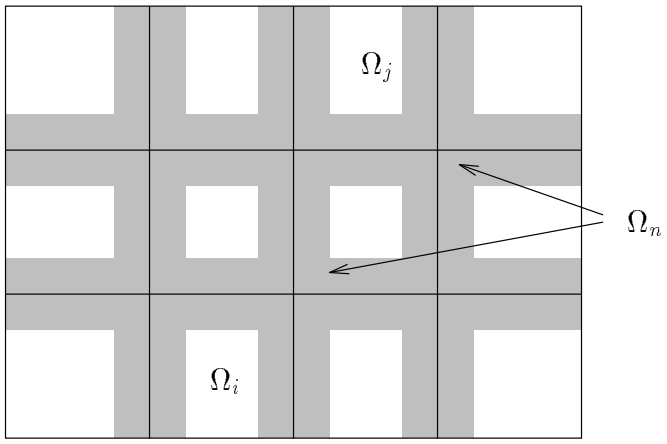


FIG. 4.3 - *Interprétation de la décomposition de la fonction de coût : les Ω_i sont délimités par les traits pleins, Ω_n est la région grisée.*

dans lesquels sont définies les variables de contrôle.

On a donc

$$J_i(x_i) = J_i(u_i, u_n) = \frac{1}{2}\int_0^T \int_{\Omega_i} \|x_i(u_i, u_n) - x_i^{obs}\|^2 d\Omega_i dt$$

où $u_i \in U_i$ pour $i \in \{1, \ldots, n\}$ et les U_i sont tels que :

$$U = \bigoplus_{i=1}^{n} U_i.$$

La fonctionnelle à minimiser peut donc se décomposer en :

$$J(u) = \sum_{i=1}^{n-1} J_i(u_i, u_n).$$

Remarque : Étant donné la forme de la relation qui lie u à x dans le cas du contrôle des conditions aux limites, c'est-à-dire que x est solution d'une équation différentielle dont u est la condition initiale, plus l'intervalle de temps sur lequel on effectue l'assimilation augmente, plus la zone qui détermine la valeur de x en un point va s'étendre, c'est-à-dire plus Ω_n va recouvrir une partie importante de Ω. On pourra même avoir $\Omega_n = \Omega$, ce qui conduira à $U_n = U$ et la fonction de coût ne se décomposera plus.

4.2.2 Relaxation de type Gauss-Seidel

Algorithme

De manière générale, on appelle méthode de Gauss-Seidel une méthode dans laquelle on résout successivement un problème par rapport à l'une de ses variables, les

autres étant fixées. Étant donnée la forme de la fonction de coût qui nous intéresse, cet algorithme va prendre une forme particulière que nous pourrons exploiter. La forme générale de ce type d'algorithme appliqué à l'optimisation est :

$$u_i^{k+1} = \arg \min_{u_i \in U_i} J(u_1^{k+1}, \ldots, u_{i-1}^{k+1}, u_i, u_{i+1}^k, \ldots, u_n^k),$$

formule que l'on itère sur i puis sur k. Avec les hypothèses que nous avons fait sur la forme de J, cet algorithme s'écrit :

$$\begin{cases} u_i^{k+1} = \arg \min_{u_i \in U_i} J_i(u_i, u_n^k) \quad \text{si} \quad 1 \leq i \leq n-1, \\ \\ u_n^{k+1} = \arg \min_{u_n \in U_n} J(u_1^{k+1}, \ldots, u_{n-1}^{k+1}, u_n). \end{cases}$$

On constate que pour $1 \leq i \leq n-1$ les problèmes d'optimisation sont indépendants et peuvent être résolus en parallèle. Chacun de ces problèmes se résout par un algorithme de contrôle tels que ceux présentés dans le chapitre 3.

Convergence

Plusieurs théorèmes peuvent s'appliquer pour montrer que l'algorithme décrit ci-dessus converge. Nous en énonçons trois et nous démontrerons le dernier d'entre eux car, même si les hypothèses qu'il suppose sont un peu plus restrictives, nous verrons qu'il peut s'adapter à d'autres cas.

Théoreme 4.1 *Si J est elliptique alors la méthode de relaxation de Gauss-Seidel converge.*

Nous rappelons qu'une fonctionnelle J est elliptique si et seulement si elle est une fois continuement dérivable sur U et si il existe une constante β telle que

$$\beta > 0 \quad \text{et} \quad (\nabla J(u) - \nabla J(v), v - u) \geq \beta \|v - u\|^2 \quad \forall u, v \in U.$$

Une démonstration de ce théorème composante par composante est donnée dans [16], p.185, elle s'adapte sans difficulté au cas « par blocs ».

Théoreme 4.2 *Si $J : \mathbb{R}^m \to \mathbb{R}$ est continuement différentiable et convexe, si pour tout i, J est une fonction strictement convexe de u_i lorsque les autres composantes sont fixées et si la suite engendrée par l'algorithme de Gauss-Seidel $\{u^k\}$ est bien définie, alors la limite de la suite $\{u^k\}$ minimise J sur $\mathbb{R}^m$.*

La démonstration de ce théorème se trouve, par exemple, dans [8], p.220. Remarquons que ce théorème ne nous assure pas de l'existence des itérés successifs.

Théoreme 4.3 *Si $J : \mathbb{R}^m \to \mathbb{R}$ est continuement différentiable, si il existe γ un réel positif tel que l'application $F : \mathbb{R}^m \to \mathbb{R}^m$ définie par $F(u) = u - \gamma \nabla J(u)$ soit contractante, alors, il existe un unique u^* qui minimise J sur $\mathbb{R}^m$, l'algorithme de Gauss-Seidel décrit ci-dessus est bien défini (c'est-à-dire qu'un minimum u_i^k existe à chaque étape) et la suite engendrée converge vers u^* de manière géométrique.*

Une démonstration de ce théorème est donnée dans [8], p.221. Nous la reprenons ici en l'adaptant au problème qui nous intéresse.

Nous rappelons le théorème du point fixe dont nous nous servirons :

Théoreme 4.4 (Théorème du point fixe) *Soit $F : I\!R^m \longrightarrow I\!R^m$ contractante de module α. Alors F possède un unique point fixe x^* sur $I\!R^m$ et pour tout $x_0 \in I\!R^m$, la suite $\{x_k\}$ engendrée par l'itération $x_{k+1} = F(x_k)$ converge vers x^* de manière géométrique, c'est-à-dire que :*

$$\|x_k - x^*\| \leq \alpha^k \|x_0 - x^*\| \ \ \forall k \geq 0.$$

Nous rappelons également qu'une fonction F est dite contractante si et seulement si il existe une constante α telle que

$$0 \leq \alpha < 1 \ \text{ et } \ \|F(u) - F(v)\| \leq \alpha \|u - v\| \ \ \forall u, v \in U.$$

Nous reprenons pour cette démonstration les notations du paragraphe précédent, l'espace dans lequel on travaille est $U = \mathbb{R}^m$, il se décompose en n sous-espaces $U_i = \mathbb{R}^{m_i}$. Tout élément $u \in U$ peut donc s'écrire $u = (u_1, \ldots, u_n)$ où $u_i \in U_i$, $1 \leq i \leq n$. Dans toute la démonstration nous utiliserons la norme dite N_∞ définie par $\|u\| = \max_{1 \leq j \leq m} |x_j|$ dans U et $\|u_i\|_i = \max_{1 \leq j \leq m_i} |(u_i)_j|$ dans U_i. On remarquera que $\|u\| = \max_{1 \leq i \leq n} \|u_i\|_i$. U étant de dimension finie, toutes les normes sont équivalentes donc nous ne restreignons pas la généralité de la démonstration.

DÉMONSTRATION : L'application $F : \mathbb{R}^m \to \mathbb{R}^m$ définie par $F(u) = u - \gamma \nabla J(u)$ est contractante par hypothèse donc on peut appliquer le théorème du point fixe et F possède un unique point fixe u^* sur $U = \mathbb{R}^m$ qui vérifie

$$u^* = F(u^*) = u^* - \gamma \nabla J(u^*),$$

donc

$$\nabla J(u^*) = 0.$$

L'équation d'Euler n'admet pas d'autre solution sinon cette solution serait point fixe de F, donc il existe un unique $u^* \in U = \mathbb{R}^m$ qui minimise J sur U.

Soit $F_i : U \longrightarrow U_i$ qui a les mêmes composantes que F sur le sous-espace U_i. Alors

$$\|F_i(u) - F_i(v)\|_i \leq \|F(u) - F(v)\| \leq \alpha \|u - v\|,$$

donc F_i est contractante. Fixons maintenant l'indice i et soit $u \in U$, on peut écrire $u = (u_i, \bar{u})$ où $u_i \in U_i$ et $\bar{u} \in U_i^{\perp}$. Pour $\bar{u}$ fixé, on définit $F_i^u : \mathbb{R}^{m_i} \longrightarrow \mathbb{R}^{m_i}$ par $F_i^u(v_i) = F_i(u_1, \ldots, u_{i-1}, v_i, u_{i+1}, \ldots, u_n), \forall v_i \in U_i$. On a alors :

$$\|F_i^u(v_i) - F_i^u(w_i)\|_i \leq \alpha \|(v_i, \bar{u}) - (w_i, \bar{u})\|$$

car F_i est contractante et

$$\|(v_i, \bar{u}) - (w_i, \bar{u})\| = \|v_i - w_i\|_i$$

car toutes les autres composantes sont égales. Donc F_i^u est contractante pour tout $u \in U$ et pour tout i. On peut appliquer le théorème du point fixe à F_i^u sur $U_i = \mathbb{R}^{m_i}$, F_i^u admet donc un unique point fixe sur U_i, c'est aussi l'unique valeur qui minimise la fonctionnelle J sur U_i, les autres composantes de u étant fixées. Ceci montre que l'algorithme proposé est bien défini, c'est à dire que

$$u_i^{k+1} = \arg \min_{u_i \in U_i} J(u_1^{k+1}, \ldots, u_{i-1}^{k+1}, u_i, u_{i+1}^k, \ldots, u_n^k)$$

existe à chaque étape.

On peut donc définir par récurrence $G_i : \mathbb{R}^m \longrightarrow \mathbb{R}^{m_i}$ qui à $u \in \mathbb{R}^m$ associe l'unique solution sur $\mathbb{R}^{m_i}$ de

$$v_i = F_i(G_1(u), \ldots, G_{i-1}(u), v_i, \ldots, u_n).$$

On définit alors l'application $G : \mathbb{R}^m \longrightarrow \mathbb{R}^m$ de composantes G_i. On a alors

$$\|G(u) - G(v)\| \leq \|G_i(u) - G_i(v)\|_i \; 1 \leq i \leq n$$

et

$$\|G_i(u) - G_i(v)\| \leq \alpha \max \left\{ \max_{j<i} \|G_j(u) - G_j(v)\|_j, \max_{j \geq i} \|u_j - v_j\|_j \right\}.$$

Une récurrence simple sur j montre que

$$\|G_j(u) - G_j(v)\|_j \leq \alpha \max_j \|u_j - v_j\|_j = \alpha \|u - v\| \;\; \forall j.$$

Donc, G est contractante et on peut appliquer le théorème du point fixe. La suite (u^k) engendrée par l'itération de G converge vers l'unique point fixe de G de manière géométrique. Or, d'après la définition de G, c'est aussi le point fixe de F donc la suite engendrée converge vers u^* de manière géométrique. Cette méthode revient à minimiser successivement $F(u_i, \bar{u})$ sur $\mathbb{R}^{m_i}$, on reconnaît l'algorithme de Gauss-Seidel. ∎

Mise en œuvre

On suppose connues les valeurs de u_i^k pour $1 \leq i \leq n$. On suppose également que les données sont distribuées de telle manière que u_i^k soit stocké sur le processeur `P_i`. Une itération de l'algorithme proposé s'écrit alors :

- `Diffuser` u_n^k,
- `Résoudre en parallèle` $u_i^{k+1} = \arg\min_{u_i \in U_i} J_i(u_i, u_n^k)$,

- `Regrouper les` u_i^{k+1} `sur` $\mathtt{P_n}$,
- `Résoudre` $u_n^{k+1} = \arg\min_{u_n \in U_n} J(u_1^{k+1}, \ldots, u_{n-1}^{k+1}, u_n)$,
- `Tester la convergence et recommencer éventuellement.`

Deux problèmes se posent alors : le coût en communication des opérations de diffusion et de regroupement et le coût en calcul de la résolution du problème $\min_{u_n \in U_n} J(u_1^{k+1}, \ldots, u_{n-1}^{k+1}, u_n)$.

Plaçons nous dans le cas de l'assimilation de données par le contrôle des conditions initiales, le domaine global Ω étant discrétisé sur une grille de taille $N \times M$. La variable de contrôle sur le domaine interface Ω_n est de taille $\mathcal{O} \times (n(N+M))$. Donc lorsque le nombre de domaines augmente, on doit diffuser et regrouper des valeurs sur un plus grand nombre de processeurs et en plus la taille des données à diffuser et regrouper augmente. On peut déjà prévoir que les performances de cet algorithme diminueront lorsque le nombre de processeurs augmentera.

Le deuxième problème qui se pose est la résolution du problème d'optimisation sur l'interface Ω_n. Ce problème est de taille $\mathcal{O}(n \times (N+M))$, elle augmente avec le nombre de processeurs. De plus, la forme du domaine est très irrégulière. On peut donc s'attendre à un temps de résolution croissant. Cette étape étant de plus non parallèle, elle va faire chuter l'efficacité de la parallélisation.

Enfin, un problème important est la forme du domaine interface. Peu de modèles sont prévus pour pouvoir être utilisés sur un tel domaine ! Il faudra donc adapter le modèle, ce qui peut demander un travail très important au niveau du codage. D'autre part, selon les méthodes numériques utilisées, un domaine aussi irrégulier peut rendre le modèle inutilisable. Le travail d'adaptation du modèle pourrait être quasiment aussi coûteux que le développement d'un nouveau modèle.

4.2.3 Relaxation de type Jacobi

Algorithme

Contrairement aux algorithmes de type Gauss-Seidel dans lesquels on résout successivement un problème par rapport à chacune de ses variables, on peut définir des algorithmes de type Jacobi dans lesquels on résoudra simultanément et indépendamment les problèmes par rapport à chacune des variables, les autres étant fixées. Les problèmes d'optimisation par rapport à chacune des variables peuvent alors évidemment être résolus en parallèle. Cela peut s'écrire sous forme d'itération :

$$u_i^{(k+1)} = \arg \min_{u_i \in U_i} J(u_1^{(k)}, \ldots, u_{i-1}^{(k)}, u_i, u_{i+1}^{(k)}, \ldots, u_n^{(k)}).$$

Dans le cas qui nous intéresse, cet algorithme s'écrit :

$$\begin{cases} u_i^{k+1} = \arg\min_{u_i \in U_i} J_i(u_i, u_n^k) \quad \text{si} \quad 1 \leq i \leq n-1, \\ \\ u_n^{k+1} = \arg\min_{u_n \in U_n} J(u_1^k, \ldots, u_{n-1}^k, u_n). \end{cases}$$

Chacun de ces problèmes se résout par un algorithme de contrôle de la même manière que dans le cadre de la méthode de type Gauss-Seidel.

Convergence

Nous avons pour cet algorithme le même résultat de convergence que pour la méthode de Gauss-Seidel. Nous le rappelons ici :

Théoreme 4.5 *Si $J : I\!R^m \to I\!R$ est continuement différentiable, si il existe γ un réel positif tel que l'application $F : I\!R^m \to I\!R^m$ définie par $F(u) = u - \gamma \nabla J(u)$ soit contractante, alors, il existe un unique u^* qui minimise J sur $I\!R^m$, l'algorithme de Jacobi décrit ci-dessus est bien défini (c'est-à-dire qu'un minimum u_i^k existe à chaque étape) et la suite engendrée converge vers u^* de manière géométrique.*

DÉMONSTRATION : Nous reprenons les notations de la démonstration de la convergence de la méthode de Gauss-Seidel. Cette démonstration reste valable pour les deux premières étapes, c'est-à-dire l'existence et l'unicité du minimum de J et l'existence et l'unicité de

$$u_i^{(k+1)} = \arg\min_{u_i \in U_i} J(u_1^{(k)}, \ldots, u_{i-1}^{(k)}, u_i, u_{i+1}^{(k)}, \ldots, u_n^{(k)})$$

à chaque étape. On peut donc définir les fonctions $H_i : \mathbb{R}^m \longrightarrow \mathbb{R}^{m_i}$ qui à $u \in \mathbb{R}^m$ associe l'unique solution de $v_i = F_i^u(v_i)$ sur $\mathbb{R}^{m_i}$ ainsi que l'application $H : \mathbb{R}^m \longrightarrow \mathbb{R}^m$ dont les composantes sont les H_i. Soient $u, v \in \mathbb{R}^m$, d'après la définition de H_i on a

$$H_i(u) = F_i(u_1, \ldots, u_{i-1}, H_i(u), u_{i+1}, \ldots, u_n),$$
$$H_i(v) = F_i(v_1, \ldots, v_{i-1}, H_i(v), v_{i+1}, \ldots, v_n).$$

Avec la norme choisie, cela donne :

$$\|H_i(u) - H_i(v)\| \leq \alpha \max \left\{ \|H_i(u) - H_i(v)\|_i, \max_{j \neq i} \|u_j - v_j\|_j \right\}.$$

Et F est contractante donc $\alpha < 1$ donc le maximum est forcément atteint dans le second terme, donc

$$\|H_i(u) - H_i(v)\| \leq \alpha \max_{j \neq i} \|u_j - v_j\|_j \leq \alpha \|u - v\|,$$

donc H est contractante. Le théorème du point fixe s'applique et nous montre que H possède un unique point fixe sur $\mathbb{R}^m$. D'après la définition de H, ce point fixe est aussi l'unique point fixe de F donc la suite engendrée converge vers u^* de manière géométrique. Cette méthode revient à minimiser simultanément $F_i^{u^k}(u_i)$ sur $\mathbb{R}^{m_i}$, on reconnaît l'algorithme de Jacobi. ∎

Mise en œuvre

L'intérêt de cet algorithme par rapport à celui de type Gauss-Seidel est qu'il ne comporte pas d'étape dissymétrique : toutes les minimisations ont lieu en même temps. Par contre, le coût de communication reste le même et comme on l'a déjà remarqué, le coût de la minimisation dans U_n augmente rapidement, les charges de calcul seront donc très déséquilibrées. Pour y remédier, on peut grouper plusieurs des autres minimisations sur un processeur, mais on se heurtera alors rapidement à des problèmes de capacité mémoire. On sait également que cette méthode converge généralement moins bien que celle de Gauss-Seidel. Les autres inconvénients de la méthode de Gauss-Seidel concernant la mise en œuvre subsistent.

4.2.4 Méthodes de gradient et de Newton

Algorithmes

L'analyse que nous allons faire dans ce paragraphe s'applique à toute une classe d'algorithmes de minimisation itératifs dont les méthodes de :

GRADIENT : on entend par là les méthodes de descente dans la direction opposée au gradient, elles diffèrent par le pas utilisé. Les principales sont :

- Gradient à pas fixe : $u_{k+1} = u_k - \rho \nabla J(u_k)$,
- Gradient à pas variable : $u_{k+1} = u_k - \rho_k \nabla J(u_k)$
- Gradient à pas optimal : $u_{k+1} = u_k - \rho(u_k) \nabla J(u_k)$ où

$$\rho(u_k) = \arg\min_{\rho \in \mathbb{R}} J(u_k - \rho \nabla J(u_k))$$

GRADIENT CONJUGUÉ : dans cette méthode, la direction de descente n'est plus le gradient de la fonctionnelle mais une combinaison de ce gradient et de la direction de descente précédente de manière à ce que les directions de descente successives soient conjuguées entre elles. Cela donne :

$$\begin{cases} d_k = \nabla J(u_k) + \dfrac{\|\nabla J(u_k)\|^2}{\|\nabla J(u_{k-1})\|^2} d_{k-1}, \\ r_k = \arg\min_{r \in \mathbb{R}} J(u_k + r d_k) = \dfrac{(\nabla J(u_{k-1}), d_k)}{(A d_k, d_k)}, \\ u_{k+1} = u_k - r_k d_k. \end{cases}$$

NEWTON : cette méthode est basée sur l'utilisation du Hessien de la fonctionnelle à minimiser qui doit être deux fois dérivable. Une itération s'écrit :

$$u_{k+1} = u_k - [\nabla^2 J(u_k)]^{-1} \nabla J(u_k).$$

La suite de ce paragraphe s'appliquera également aux méthodes de Newton généralisées, c'est-à-dire dans lesquelles on se contente d'approximer le Hessien par une matrice plus simple à inverser.

Convergence

Les démonstrations de convergence de ces méthodes sont classiques, on peut les trouver par exemple dans [16], chapitres 7 et 8.

Parallélisation

Un point commun à toutes ces méthodes est que l'on a besoin de connaître la valeur du gradient de la fonctionnelle à minimiser au point courant pour calculer l'itéré suivant. Dans le cadre de l'assimilation de données, on a vu au chapitre 3 que l'on dispose du modèle adjoint pour calculer ce gradient (ainsi que du modèle adjoint du second ordre pour calculer le Hessien dans le cas de la méthode de Newton). Le gradient est donc obtenu par intégration rétrograde d'une équation d'évolution que l'on peut paralléliser. Dans le cas de la méthode de Newton nous retrouvons l'algorithme du paragraphe 4.1.

Le reste de ces algorithmes est essentiellement constitué de mises à jour de vecteurs et de calculs de produits scalaires. Comme nous l'avons déjà vu au paragraphe 4.1, leur coût est très faible par rapport à celui du calcul du gradient et leur parallélisation est relativement peu efficace. Cela conduira à de mauvaises performances si le coût de cette partie du code devient non négligeable par rapport à l'intégration parallèle des modèles, ce qui ne sera vraisemblablement jamais le cas avec un modèle d'atmosphère ou d'océan réaliste, nous étudierons ce cas plus en détail au chapitre 5.1.

4.2.5 Méthodes de relaxation asynchrones

Méthodes asynchrones

Un inconvénient des méthodes de relaxation définies jusque là est qu'elles sont synchrones, c'est-à-dire que tous les processeurs effectuent en même temps la même itération. Or, certains peuvent résoudre le problème local qu'ils ont à traiter plus vite que les autres. On peut donc envisager des méthodes dans lesquelles on commence sans attendre l'itération suivante avec les informations disponibles. Cela introduit évidemment un décalage des processeurs en nombre d'itérations d'où l'appellation « asynchrone » de ce type de méthode. L'intérêt de cette approche en parallélisme est que les processeurs sont toujours actifs, la difficulté est l'inutilité éventuelle de cette activité.

Algorithme général de relaxation asynchrone

On peut généraliser les méthodes de Gauss-Seidel et de Jacobi vues dans ce chapitre en définissant des versions asynchrones de ces algorithmes. Autrement dit, on peut laisser un processeur qui, pour une raison quelconque, a terminé une itération en commencer une autre avec les valeurs les plus récentes dont il dispose en

provenance des autres processeurs. On peut l'écrire sous la forme :

$$u_i^{k_i+1} = \arg\min_{u_i \in U_i} J(u_1^{k_1}, \ldots, u_{i-1}^{k_{i-1}}, u_i, u_{i+1}^{k_{i+1}}, \ldots, u_n^{k_n}).$$

La notation $u_j^{k_j}$ représente la dernière valeur connue de la variable u_j. Elle dépend du nombre d'itérations effectuées par le processeur j mais aussi du temps de communication entre les processeurs i et j.

Convergence

On peut montrer que ce type d'algorithme appliqué à la recherche d'un point fixe d'une fonction converge sous des hypothèses assez restrictives (voir [8], chapitre 6). On montre notamment que l'algorithme de gradient asynchrone pour la minimisation d'une fonctionnelle converge si le Hessien de cette fonctionnelle est à diagonale dominante.

On peut montrer que cet algorithme converge avec les mêmes hypothèses que pour les démonstrations de convergence des algorithmes de Gauss-Seidel et Jacobi si on ajoute une condition d'asynchronisme partiel, c'est-à-dire si on borne l'écart maximal en nombre d'itérations entre deux processeurs (voir [8], chapitre 7, partie 7.5).

Application à l'assimilation de données

Dans l'application qui nous préoccupe, la fonctionnelle à minimiser peut se décomposer en :

$$J(u) = \sum_{i=1}^{n-1} J_i(u_i, u_n)$$

où $u_i \in U_i$ pour $i \in \{0, \ldots, n\}$. Nous pouvons appliquer l'algorithme précédent à cette fonctionnelle, on obtient :

$$\begin{cases} u_i^{k+1} = \arg\min_{u_i \in U_i} J_i(u_i, u_n^{k_n}) \quad \text{si} \quad 1 \leq i \leq n-1, \\ \\ u_n^{k+1} = \arg\min_{u_n \in U_n} J(u_1^{k_1}, \ldots, u_{n-1}^{k_{n-1}}, u_n). \end{cases}$$

Là encore, chacun des problèmes sera résolu par un algorithme de contrôle tels que ceux présentés dans le chapitre 3.

Mise en œuvre

Nous allons retrouver les mêmes problèmes pour appliquer cette méthode que ceux rencontrés pour les algorithmes de relaxation classiques. La principale difficulté est due à la dissymétrie entre les domaines. Il semble que cette version apporte une réponse au problème du déséquilibre de charge mais en fait, l'écart entre le nombre

d'itérations sur le domaine «interface» et les autres risque d'augmenter très vite. Sachant qu'un bon algorithme d'assimilation de données converge après une dizaine d'itérations, on risque de faire beaucoup d'itérations pour rien sur les domaines réguliers avant d'avoir fait quelques itérations sur l'interface. Cette approche ne semble donc pas très intéressante dans le cadre de l'assimilation de données.

4.3 Utilisation de la trajectoire

4.3.1 Contrôle de la variable d'état

De la même manière qu'au paragraphe 4.2.1, on considère J comme une fonction de la variable d'état. On a vu qu'elle peut alors s'écrire :

$$J(x) = \sum_{i=1}^{n-1} J_i(x_i) \quad \text{où} \quad J_i(x_i) = \frac{1}{2}\int_0^T \int_{\Omega_i} \|x_i - x_i^{obs}\|^2 d\Omega_i dt.$$

On notera V l'espace des fonctions du domaine d'assimilation Ω à valeurs dans $\mathbb{R}$. Soit H le sous-ensemble de V constitué des valeurs possibles de la variable d'état du système. Le problème d'assimilation des données s'écrit alors :

$$\text{Trouver} \quad x^* \in H \quad \text{tel que} \quad J(x^*) = \min_{x \in H} J(x).$$

D'autre part, soit H_i l'ensemble des restrictions sur Ω_i des éléments de H. Considérons alors le problème :

$$\begin{cases} \text{Trouver} \quad \bar{x}_i \in H_i \quad \text{tel que} \quad J_i(\bar{x}_i) = \min\limits_{x_i \in H_i} J_i(x_i) \; \forall i \in \{1, \ldots, n\} \\ \\ \bar{x} = (\bar{x}_1, \ldots, \bar{x}_n) \in H. \end{cases}$$

Remarque : $\bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ signifie que $\bar{x}$ est défini sur Ω par $\bar{x} = \bar{x}_i$ sur Ω_i pour tout i.

Alors

$$J_i(\bar{x}_i) \leq J_i(x_i^*), \; \forall i \in \{1, \ldots, n\}.$$

Comme on a imposé la contrainte $\bar{x} \in H$, on a :

$$\sum_{i=1}^{n} J_i(\bar{x}_i) = J(\bar{x})$$

et

$$J(\bar{x}) \leq J(x), \; \forall x \in H.$$

donc $\bar{x}$ est donc solution du problème initial.

Les deux problèmes que nous venons de formuler sont équivalents. Nous pouvons donc, dans ce cas simple, décomposer un problème d'optimisation en un ensemble de problèmes couplés de taille plus petite avec contraintes.

4.3.2 Contrôle réaliste

Dans les problèmes qui nous intéressent en assimilation de données, l'hypothèse selon laquelle on peut décomposer la fonction de coût en fonctions locales n'est pas vérifiée. En effet, dans ce type de problèmes, la fonction de coût doit être considérée comme une fonction des variables de contrôle et non des variables d'état. On peut généralement mettre la fonction de coût sous la forme :

$$J(u) = \frac{1}{2}\int_0^T \int_\Omega \|x(u) - x^{obs}\|^2 d\Omega dt.$$

La variable d'état x dépend de la variable de contrôle par l'intermédiaire d'un système (représentant un modèle atmosphérique par exemple) qui n'est ni linéaire ni local. Pour une meilleure estimation de l'erreur commise, on doit également introduire dans la fonction de coût la matrice de covariance d'erreur C. La fonction de coût s'écrit alors :

$$J(u) = \frac{1}{2}\int_0^T \int_\Omega \|C.x(u) - x^{obs}\|^2 d\Omega dt.$$

Cette matrice étant en théorie pleine, on a un deuxième facteur de non localité de la fonction de coût, c'est-à-dire un deuxième facteur qui nous empêche d'écrire l'intégrale sous la forme d'une somme d'intégrales sur des sous-domaines.

On peut essayer de se ramener à un ensemble de problèmes avec contraintes comme dans le cas précédent. La contrainte doit intégrer tout ce qui n'est pas local à un sous-domaine, elle devient de ce fait très difficile à exprimer.

4.3.3 Application

On peut garder la fonction de coût comme une fonction de la variable d'état. On inclura alors dans les contraintes le fait qu'elle doit être solution du modèle. Cela donne la formulation suivante :

$$\begin{cases} \text{Trouver} \quad x^* \in H \quad \text{tel que} \quad J(x^*) = \min\limits_{x \in H} J(x), \\ \\ H = \left\{ x \in V \quad \text{tel que} \quad \dfrac{dx}{dt} = F(x) \right\} \end{cases}$$

Sous cette forme, on peut décomposer l'intégrale, le problème s'écrit alors :

$$\begin{cases} \text{Trouver} \quad x_i^* \in H_i \quad \text{tel que} \quad J(x_i^*) = \min\limits_{x_i \in H_i} J_i(x_i), \\ \\ H_i = \left\{ x_i \in V_i \quad \text{tel que} \quad \dfrac{dx_i}{dt} = F_i(x_1, \ldots, x_i, \ldots, x_n) \right\} \end{cases}$$

La contrainte peut aussi s'écrire $\varphi_i(x) = 0$ avec

$$\varphi_i(x) = \frac{dx_i}{dt} - F_i(x_1, \ldots, x_i, \ldots, x_n).$$

Les problèmes qui se posent alors sont des problèmes d'optimisation avec contraintes que l'on peut traiter avec des algorithmes différents de ceux rencontrés jusque là. On peut par exemple utiliser les algorithmes suivants :

PÉNALITÉ : On peut formuler chacun de ces problèmes sous une forme pénalisée : on cherche x_ε^* qui minimise la fonctionnelle

$$J_i^\varepsilon(x_i) = J_i(x_i) + \frac{1}{\varepsilon}\|\varphi_i(x_1, \ldots, x_n)\|,$$

et on a $\lim_{\varepsilon \longrightarrow 0} x_\varepsilon^* = x^*$.

DUALITÉ : On peut aussi formuler ces problèmes sous forme de recherche du point selle de :

$$L(x, p) = \sum_{i=1}^{n} J_i(x_i) + \left(p, \sum_{i=1}^{n} \varphi_i(x_1, \ldots, x_n)\right).$$

Ce problème se résout alors par un algorithme tel que celui d'Uzawa.

LAGRANGIEN AUGMENTÉ : Une autre approche possible est l'utilisation de la méthode du Lagrangien augmenté qui consiste à résoudre un problème de point selle pour

$$L(x, p) = \sum_{i=1}^{n} J_i(x_i) + \left(p, \sum_{i=1}^{n} \varphi_i(x_1, \ldots, x_n)\right) + \frac{c}{2}\|\varphi_i(x_1, \ldots, x_n)\|^2.$$

Un problème important de cette approche est le calcul de φ_i avec les $(x_j)_{j \neq i}$ connus. Étant donné la forme de φ_i, il faut intégrer le modèle utilisé pour connaître la valeur de cette fonction. Ce calcul devra être fait sur le domaine complet et sera dans la plupart des cas beaucoup trop coûteux pour pouvoir être répété sur tous les processeurs à chaque pas de la minimisation. On a décomposé la fonction de coût mais la contrainte reste globale. Cette approche n'est donc pas intéressante dans le cadre du calcul parallèle.

4.4 Couplage et assimilation

4.4.1 Couplage de modèles

Un problème qui se pose actuellement pour faire une bonne assimilation de données est qu'un fluide géophysique ne peut pas être considéré comme un milieu isolé. La prévision de son évolution requiert donc la connaissance de l'évolution de ces interfaces avec les milieux environnants. De nombreux travaux de recherche sont actuellement en cours dans le domaine du couplage de modèles pour mieux prendre en compte ces phénomènes. Par modèles couplés on entend :

- Couplage entre plusieurs milieux physiques distincts comme, par exemple, l'océan et l'atmosphère ou l'atmosphère et le sol.

- Couplage entre deux modèles de même nature (météorologiques par exemple) mais d'échelles différentes. On peut coupler un modèle global avec un modèle local pour faire de la prévision avec une meilleure précision sur une région donnée et prendre en compte l'influence des phénomènes de grande échelle.
- On peut aussi coupler des modèles de même résolution mais adaptés à des régions voisines de caractéristiques différentes comme une région sur la mer et une région terrestre.
- On peut coupler des modèles qui s'appliquent sur le même domaine mais qui sont de nature différente comme un modèle météorologique et un modèle chimique. Cela permet d'étudier par exemple la dispersion d'un produit polluant et son effet sur l'atmosphère (étude de l'ozone).
- On peut aussi considérer l'utilisation d'un même modèle sur plusieurs domaines contigus, c'est-à-dire la décomposition de domaine comme une forme de couplage.
- Une combinaison de ces différents types de couplages.

L'interface peut être géométrique comme c'est le cas de l'interface entre l'océan et l'atmosphère (dans ce cas, il faut prendre en compte les échanges d'énergie et d'humidité) ou dans le cas de l'interface entre deux modèles spécifiques d'un même milieu adapté à des régions voisines. Mais, elle peut aussi être plus abstraite, on peut par exemple considérer le couplage d'un modèle météorologique avec un modèle de chimie pour étudier la pollution. Dans ce cas, l'interface est constituée par les échanges d'énergie. Quelle que soit l'interface, le couplage de modèle doit être réalisé pour obtenir une prévision plus fine.

Nous proposons ici de considérer la méthode de décomposition de domaines comme un cas particulier (simple) du couplage de modèle. Nous avons vu au chapitre 3 que les méthodes adjointes nous permettent de contrôler les conditions initiales mais aussi tout autre paramètre d'un modèle. Nous pouvons donc contrôler ses «interfaces» avec un autre modèle.

4.4.2 Contrôle des interfaces

On a vu au paragraphe 3.4.6 que l'on peut utiliser le même modèle adjoint pour contrôler les conditions initiales d'un modèle mais aussi n'importe lequel de ses paramètres. Nous pouvons donc appliquer cette technique pour contrôler les «interfaces» de ce modèle.

On suppose que l'évolution de l'atmosphère est donnée par :

$$\begin{cases} \dfrac{dx}{dt} = F(x, v) \\ x(0) = u \end{cases}$$

où x représente la variable d'état du modèle , u est la condition initiale et v la condition aux interfaces du modèle. v peut représenter des grandeurs très diverses. Dans le cas d'un modèle atmosphérique couplé avec un modèle d'océan, v peut représenter la température de l'eau à la surface de l'océan telle qu'elle est fournie par le modèle d'océan, dans le cas inverse, il peut représenter la vitesse du vent prévue par le modèle météorologique à la surface de l'océan. v peut aussi être un champ de température que l'on passe d'un modèle météorologique à un modèle chimique. Ce paramètre peut aussi représenter une condition aux limites dans le cas d'une décomposition de domaines. On cherche les conditions u et v minimisant :

$$J(u,v) = \frac{1}{2}\int_0^T \|C.x(u,v) - x^{obs}\|_\Omega^2 dt$$

Pour une perturbation h, le système linéaire tangent s'écrit :

$$\begin{cases} \dfrac{d\hat{x}}{dt} = \left[\dfrac{\partial F}{\partial x}\right].\hat{x} + \left[\dfrac{\partial F}{\partial v}\right].h_v \\ \\ \hat{x}(0) = h_u \end{cases}$$

Soit p solution du système adjoint :

$$\begin{cases} \dfrac{dp(t)}{dt} + \left[\dfrac{\partial F}{\partial x}\right]^*.p(t) = C^*.\left(C.x(t) - x^{obs}(t)\right) \\ \\ p(t) = 0 \end{cases}$$

En multipliant cette équation par $\hat{x}(t)$ et en intégrant par parties, on obtient :

$$-\left(p(0), h_u\right) - \left(\left[\frac{\partial F}{\partial v}\right]^* p, h_v\right) = \hat{J}(u,v,h)$$

Donc les composantes du gradient sont :

$$\begin{cases} \nabla_u J = -p(0) \\ \\ \nabla_v J = -\left[\dfrac{\partial F}{\partial v}\right]^*.p \end{cases}$$

Le calcul est le même qu'au paragraphe 3.4.6. $\nabla_v J$ est utilisé pour optimiser le choix des conditions aux interfaces v. Le système adjoint reste le même que dans les autres cas déjà rencontrés, on applique un opérateur différent aux variables adjointes, ici cet opérateur est :

$$\left[\frac{\partial F}{\partial v}\right]^*.$$

Cette méthode peut donc s'appliquer quelles que soit les interfaces qui nous intéressent, la seule modification à apporter sera de changer l'opérateur à appliquer à la variable adjointe pour obtenir le gradient.

4.4.3 Assimilation de données pour modèles couplés

Nous avons vu comment contrôler les interfaces d'un modèle donné. Voyons comment appliquer ceci au cas de plusieurs modèles couplés qui interagissent. Considérons un ensemble de modèles :

$$\begin{cases} \dfrac{dx_i}{dt} = F_i(x_i, v_{ij}) \\ \\ x_i(0) = u_i \end{cases}$$

où, pour $i \in \{1, \cdots, N\}$, x_i est la variable d'état pour le modèle i et v_{ij} est la variable d'état du modèle i à l'interface entre les modèles i et j. La fonction de coût pour l'assimilation de données associée à chaque modèle peut alors être définie par :

$$J_i(u_i, v_i) = \frac{1}{2} \int_0^T \|C_i.x_i(u_i, v_i) - x_i^{obs}\|^2_{\Omega_i}.$$

On peut réaliser l'assimilation de données sur ces modèles en parallèle puisqu'ils sont, pour l'instant, indépendants. Nous devons maintenant réaliser le couplage entre ces modèles. Pour cela, nous proposons d'ajouter à la fonction de coût un terme de couplage qui mesure l'écart entre les modèles aux interfaces :

$$J_i'(v_i) = \frac{1}{2} \sum_{j \neq i} \int_0^T \|C_{ij}.v_i - C_{ji}.v_j\|^2_{\Gamma_{ij}}$$

où C_{ij} et C_{ji} représentent des opérateurs qui permettent de passer des variables d'état aux interfaces à un espace convenable pour mesurer l'écart entre les modèles. Dans ce coût, le terme $C_{ji}.v_j$ est une donnée qui provient du modèle j, on contrôle le terme $C_{ij}.v_i$. La fonction de coût totale pour le modèle i est donc :

$$J_i(u_i, v_i) = \frac{1}{2} \left[\int_0^T \|C_i.x_i - x_i^{obs}\|^2_{\Omega_i} + \sum_{j \neq i} \int_0^T \|C_{ij}.v_i - C_{ji}.v_j\|^2_{\Gamma_{ij}} \right].$$

Cette modification de la fonction de coût entraîne un changement du second membre du système adjoint qui devient :

$$\begin{cases} \dfrac{dp_i}{dt} + \left[\dfrac{\partial F_i}{\partial x_i}\right]^* .p_i = C_i^* \left(C_i x_i - x_i^{obs}\right) + C_{ij}^* \left(C_{ij}.v_i - C_{ji}.v_j\right) \\ \\ p_i(T) = 0 \end{cases}$$

Et on a toujours

$$\nabla_{v_i} J_i = - \left[\frac{\partial F_i}{\partial x_i}\right]^* .p_i.$$

Le terme que l'on ajoute à la fonction de coût peut prendre différentes formes selon la relation qui lie les modèles aux interfaces. Cela peut être une relation de continuité des champs, de conservation d'énergie, etc... mais le principe reste le même.

4.4.4 Cas de la décomposition de domaine

Dans le cadre général du couplage de modèles, on travaille avec des modèles différents. On peut minimiser l'écart aux interfaces de la même manière que l'écart entre la solution contrôlée et l'observation car il est impossible d'avoir une égalité stricte aux interfaces. Par contre, dans le cas de la décomposition de domaine, on peut imposer comme contrainte que les interfaces coïncident exactement, ce qui s'exprime par une relation du type :

$$\begin{cases} \text{Trouver} \quad (u_i^*, v_i^*) = \arg\min_{u,v} J_i(u,v), \\ \\ \varphi_{ij}(v_{ij}, v_{ji}) = 0, \end{cases}$$

où φ_{ij} est une fonction qui mesure l'écart aux interfaces entre les modèles.

On retrouve ainsi une formulation proche de celle obtenue au paragraphe 4.3. L'expression de la fonction φ_{ij} est moins complexe que celle rencontrée alors mais elle n'est toujours pas locale. De plus, on doit résoudre une série de problèmes avec contraintes, ce qui nécessite sur chaque domaine la connaissance des v_{ji} qui évoluent. On utilisera donc un algorithme itératif pour résoudre un problème à « contraintes variables », ce qui pose des problèmes de convergence non résolus.

4.5 Algorithme d'assimilations couplées

Un algorithme parallèle consiste donc en une itération du processus d'assimilation de données dans chaque sous-domaine avec un échange des conditions aux limites entre deux itérations. L'algorithme est :

1. Choisir une condition initiale «à la main».
2. Intégrer le modèle direct de 0 à T, stocker la trajectoire (donne J_n).
3. Intégrer le modèle adjoint de T à 0 (intégration rétrograde, donne ∇J_n).
4. Si $\|\nabla J(u_n)\| \leq \varepsilon$ arrêter.
5. Calculer une direction de descente D_n (Newton, gradient conjugué, ...).
6. Calculer le pas de descente ρ_n tel que :

$$J(u_n + \rho_n D_n) = \min_{\rho} J(u_n + \rho D_n)$$

7. Mettre à jour la condition initiale : $u_{n+1} = u_n + \rho_n D_n$.
8. **Échanger** les valeurs aux interfaces avec les domaines voisins.
9. Itérer.

Un intérêt de cet algorithme est qu'il demande peu de modifications pour obtenir un code parallèle à partir d'un code séquentiel existant. Ces modifications consistent à modifier la fonction de coût et à ajouter l'échange des valeurs aux interfaces entre deux itérations de l'algorithme de minimisation.

4.6 Conclusion

Nous avons, dans ce chapitre, passé en revue les différentes approches possibles pour paralléliser le problème de l'assimilation de donnée variationnelle. Nous avons distingué trois classes de parallélisation possibles.

La première consiste à utiliser des modèles directs et adjoints parallèles dans un algorithme classique. Elle devrait donner de bons résultats si les modèles se parallélisent bien et si la phase de minimisation proprement dite reste négligeable (en temps d'exécution) par rapport à l'intégration des modèles. Un avantage majeur de cette approche est la réutilisation possible du modèle parallélisé pour la prévision elle-même.

La deuxième approche nous a conduit à une décomposition de domaine incluant un *domaine interface* à partir de laquelle nous pouvons définir des méthodes de relaxation de type Gauss-Seidel, Jacobi ou asynchrones. Deux inconvénients importants ressortent de cette étude : la définition compliquée du *domaine interface* peut poser de sérieux problèmes, à la fois sur le plan numérique et informatique, pour intégrer les modèles directs et adjoints. D'autre part, l'algorithme construit à partir de cette approche est naturellement déséquilibré en calculs.

La dernière approche consiste à décomposer le domaine d'assimilation, à considérer sur chaque domaine une restriction du modèle et à coupler les domaines par le biais du contrôle des conditions aux limites de chaque sous-domaine. Le principal avantage de cette approche est qu'elle se généralise de manière immédiate pour s'appliquer au problème de l'assimilation de données pour des modèles couplés. Elle ouvre donc la porte à une nouvelle classe d'applications de plus en plus importantes sur le plan pratique.

Chapitre 5

Application au modèle de Shallow Water

Dans ce chapitre, nous présentons l'application des méthodes de parallélisation du chapitre précédent au modèle de Shallow water. Nous commencerons par une courte présentation de ce modèle puis nous étudierons sa parallélisation ainsi que celle de son code adjoint et les performances obtenues. Nous présenterons ensuite plusieurs algorithmes parallèles d'assimilation de données et comparerons leurs performances.

5.1 Le modèle de Shallow-Water

Ce modèle est largement utilisé en météorologie et en océanographie car il tient compte de la plupart des degrés de liberté physiques (y compris les ondes gravitationnelles) présents dans les modèles tridimensionnels plus sophistiqués. Il est moins coûteux en calcul et on peut espérer que son comportement sera similaire à celui de ces modèles, c'est pourquoi il est souvent utilisé pour tester de nouvelles méthodes avant de les appliquer à un système plus complexe.

5.1.1 Les équations de Shallow-Water

Le système des équations du modèle de Shallow-Water est un système non linéaire d'équations aux dérivées partielles du premier ordre. C'est un modèle bidimensionnel d'évolution de l'atmosphère ou de l'océan. Il suppose que le fluide considéré est homogène, incompressible et sans viscosité. Les équations peuvent s'écrire sous la forme:

$$\begin{cases} \dfrac{\partial u}{\partial t} + u\dfrac{\partial u}{\partial x} + v\dfrac{\partial u}{\partial y} - fv + \dfrac{\partial \Phi}{\partial x} = 0 \\ \dfrac{\partial v}{\partial t} + u\dfrac{\partial v}{\partial x} + v\dfrac{\partial v}{\partial y} + fu + \dfrac{\partial \Phi}{\partial y} = 0 \\ \dfrac{\partial \Phi}{\partial t} + u\dfrac{\partial \Phi}{\partial x} + v\dfrac{\partial \Phi}{\partial y} + \Phi\left(\dfrac{\partial u}{\partial x} + \dfrac{\partial v}{\partial y}\right) = 0 \end{cases} \tag{5.1}$$

où f est le paramètre représentant la force de Coriolis, u et v sont les composantes du vent et Φ est le géopotentiel. Tous les calculs sont discrétisés en espace par un schéma de différences finies centrées et un schéma explicite «saute-mouton» en temps.

5.1.2 Observations

Pour mener à bien tous les essais décrits dans ce chapitre, nous utiliserons des observations synthétiques. Ces observations seront engendrées par le modèle décrit ci-dessus. Les conditions initiales utilisées pour cela seront données par la formule de Grammeltvedt :

$$h = h_0 + h_1 \tanh\left(\frac{9(y-y_0)}{2D}\right) + h_2 \sinh\left(\frac{9(y-y_0)}{D}\right) \sin\left(\frac{2\pi x}{L}\right)$$

avec $\Phi = gh$, $h_0 = 2000\,\mathrm{m}$, $h_1 = -220\,\mathrm{m}$, $h_2 = 133\,\mathrm{m}$, $g = 10\,\mathrm{m.s^{-2}}$ et $f = 10^{-4}\,\mathrm{s^{-1}}$. L et D sont les dimensions du domaine de calcul, $y_0 = D/2$ est le milieu du domaine. Les champs de vitesse initiaux sont dérivés du champ de hauteur initial par la relation géostrophique :

$$u = -\frac{g}{f}\frac{\partial h}{\partial y}$$

$$v = \frac{g}{f}\frac{\partial h}{\partial x}$$

Les conditions aux limites seront périodiques dans la direction est-ouest et à composante normale du vent nulle au nord et au sud.

Le résultat de l'intégration du modèle dans ces conditions sera stocké et tiendra lieu d'observations. Le point de départ des différentes assimilations effectuées sera obtenu en perturbant aléatoirement les conditions initiales et les conditions aux limites dans une proportion de 10 % sur les composantes du vent et le géopotentiel.

5.1.3 Le problème test

Le domaine physique considéré est de 6000 km sur 4400 km. La fonction de coût sera :

$$J = W_u \sum \left(U^{obs} - U\right)^2 + W_v \sum \left(V^{obs} - V\right)^2 + W_\Phi \sum \left(\Phi^{obs} - \Phi\right)^2$$

où U^{obs}, V^{obs}, et Φ^{obs} sont les valeurs des observations, W_u, W_v, W_Φ sont des coefficients de pondération de valeurs respectives $10^{-2}, 10^{-2}$ et 10^{-6}. Ces coefficients peuvent dépendre du point et de l'instant considéré, ils représentent alors la confiance que l'on accorde à une observation. Le critère d'arrêt pour tous les essais présentés sera :

$$\|g_k\| \leq 10^{-4} \times \|g_0\|$$

où g_k est le gradient après k itérations et g_0 le gradient initial.

5.1.4 Implémentation

Tous les programmes parallèles testés dans ce chapitre ont étés écrits en Fortran 77 standard et utilisent la bibliothèque PVM (voir chapitre 2) pour les échanges de messages. L'avantage principal de cette solution est la portabilité du code. Cela permet aussi d'atteindre de bonnes performances puisque les compilateurs Fortran sont très efficaces sur toutes les machines et que les constructeurs de machines parallèles proposent depuis peu des versions optimisées de PVM pour leurs machines (voir également chapitre 2).

En Fortran 77, les choix de discrétisation précisés plus haut pour le modèle de Shallow water donnent le code suivant :

```
DO J=2,MM1
  DO I=2,NM1
    IP1=I+1
    IM1=I-1
    JP1=J+1
    JM1=J-1

    AA= (U(IP1,J,K)-U(IM1,J,K))*(U(IP1,J,K)-U(IM1,J,K))*HX
    BB= (V(I,JP1,K)-V(I,JM1,K))*(V(I,JP1,K)-V(I,JM1,K))*HY
    CC= (U(IP1,J,K)+U(IM1,J,K))*(V(IP1,J,K)-V(IM1,J,K))*HX
    DD= (V(I,JP1,K)+V(I,JM1,K))*(V(I,JP1,K)-V(I,JM1,K))*HY

    PHIX= (PHI(IP1,J,K)-PHI(IM1,J,K))*HHX
    PHIY= (PHI(I,JP1,K)-PHI(I,JM1,K))*HHY

    FU= F(I,J)*U(I,J,K)
    FV= F(I,J)*V(I,J,K)

    PHIU= (PHI(IP1,J,K)*U(IP1,J,K)-PHI(IM1,J,K)*U(IM1,J,K))*HHX
    PHIV= (PHI(I,JP1,K)*V(I,JP1,K)-PHI(I,JM1,K)*V(I,JM1,K))*HHY

    U(I,J,KFUTURE)=   U(I,J,KPASS)   -DT*(AA+BB+PHIX-FV)
    V(I,J,KFUTURE)=   V(I,J,KPASS)   -DT*(CC+DD+PHIY+FU)
    PHI(I,J,KFUTURE)= PHI(I,J,KPASS) -DT*(PHIU+PHIV)
  ENDDO
ENDDO
```

Le coût de calcul correspondant à cette implémentation est de 26 additions et 21 multiplications par point de discrétisation. Il faut y ajouter le calcul de la fonction de coût, c'est-à-dire 9 additions et 6 multiplications par point. On arrive donc à un total de 62 opérations par point. Le coût de calcul du système adjoint est de 75 additions et 79 multiplications soit 154 opérations flottantes par point.

5.2 Parallélisation directe

Dans ce premier essai de parallélisation, nous étudions les performances que l'on peut obtenir en utilisant un modèle parallèle dans un algorithme d'assimilation de données; c'est-à-dire que l'algorithme d'assimilation de données est classique, on remplace seulement le code du modèle météorologique et son adjoint par leur équivalent parallèle. En d'autres termes, on parallélise le calcul de la fonction de coût et de ses dérivées.

5.2.1 Parallélisation du modèle

Nous parallélisons le modèle décrit au paragraphe 5.1 en utilisant une décomposition de domaine dans les deux directions est-ouest et nord-sud. On distribue alors chaque sous-domaine à un processeur. À chaque pas de temps, les processeurs chargés de sous-domaines voisins devront échanger les valeurs des champs aux interfaces pour continuer l'intégration du modèle. L'algorithme consiste donc simplement à itérer les deux étapes décrites ci-dessous :

```
Lire les conditions initiales,
Itérer sur le nombre de pas de temps :
   Calculer les valeurs des champs,
   Échanger les valeurs aux interfaces avec les sous-domaines
   voisins.
```

Pour améliorer les performances de la parallélisation, on peut commencer le calcul par les valeurs des champs aux interfaces et les envoyer. Pendant le temps de propagation des messages, on effectue le calcul à l'intérieur des sous-domaines, à la fin du calcul, on peut traiter les messages provenant des processeurs voisins qui sont arrivés entre temps. On peut ainsi espérer masquer le temps de propagation des messages. L'algorithme correspondant peut s'écrire :

```
Lire les conditions initiales,
Itérer sur le nombre de pas de temps :
   Calculer les valeurs des champs aux interfaces,
   Envoyer les valeurs aux interfaces aux sous-domaines voisins,
   Calculer les valeurs des champs à l'intérieur du sous-domaine,
   Recevoir les valeurs aux interfaces des sous-domaines voisins.
```

On parallélise alors les codes linéaire tangent et adjoints au premier et second ordre en utilisant la méthode décrite au paragraphe 3.5.7.

5.2.2 Performances des modèles

Les performances obtenues par cette méthode sont données sur les figures 5.1 et 5.2. On constate plusieurs phénomènes sur ces figures. Tout d'abord, on obtient une meilleure efficacité pour l'intégration parallèle du modèle de Shallow water et de

son adjoint sur le Cray T3D que sur l'IBM SP1. Ensuite, sur ces deux machines, les performances obtenues lors de l'intégration du modèle adjoint sont meilleures que pour l'intégration du modèle direct. Enfin, on constate que, dans tous les cas, le recouvrement des communications par le calcul n'apporte pas d'amélioration sensible.

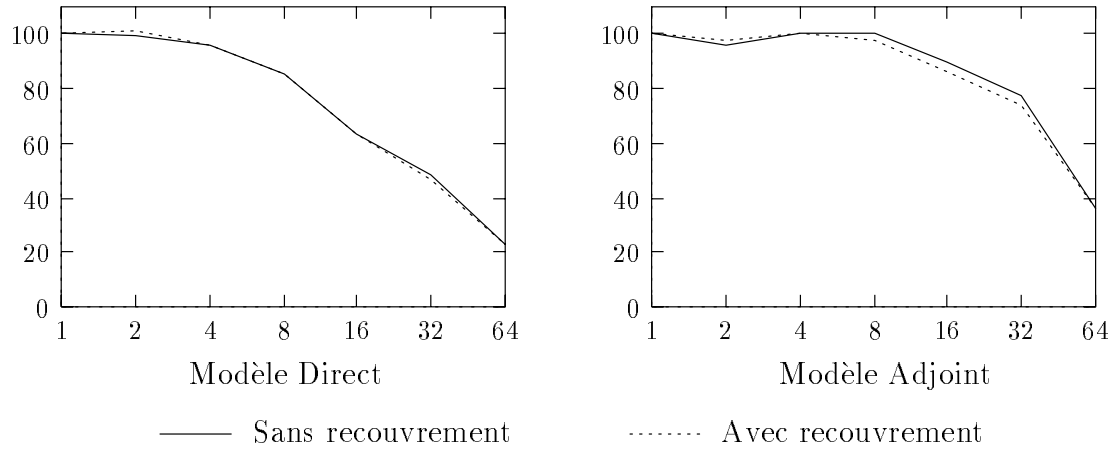


FIG. 5.1 - *Efficacité en fonction du nombre de processeurs sur T3D.*

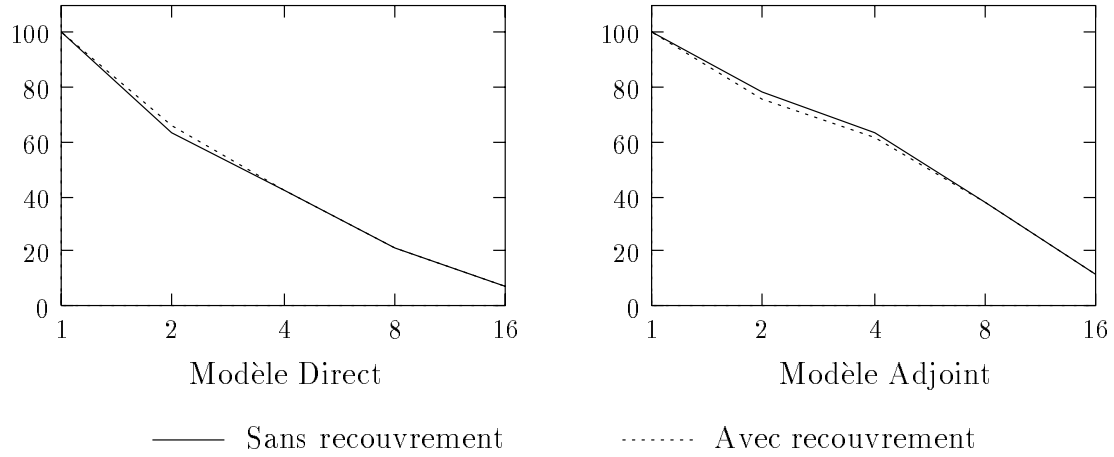


FIG. 5.2 - *Efficacité en fonction du nombre de processeurs sur SP1.*

Les résultats obtenus sur le Cray sont meilleurs car le rapport entre la vitesse de calcul et la vitesse de communication est plus faible. Il faut prendre garde à ces résultats d'efficacité qui ne reflètent pas complètement les performances de ces deux machines. Les temps d'exécution sont sensiblement les mêmes sur les deux machines avec 4 processeurs. L'intégration est plus rapide sur le SP1 avec moins de processeur car la vitesse de calcul devient prépondérante, elle est plus rapide sur le T3D avec 8 processeurs ou plus car alors la vitesse de communication devient le facteur prépondérant (voir tableau 5.1). On peut noter ici que les temps d'exécution sur l'IBM

SP1 présentent de très fortes variations d'une exécution à l'autre, cela est dû aux perturbations induites par le système d'exploitation de la machine qui comporte un système UNIX complet par processeur. Ce phénomène n'apparaît pas sur le Cray T3D car le système d'exploitation sur chaque nœud a été réduit au minimum (micro-noyau).

Nombre de Processeurs	SP1		T3D	
	Direct	Adjoint	Direct	Adjoint
1	800	1850	1420	3440
2	610	1220	700	1750
4	480	740	370	800
8	480	610	210	430
16	730	1030	140	240

TAB. 5.1 - *Meilleur temps d'exécution sur SP1 et T3D en ms*

L'efficacité obtenue est meilleure dans le cas du modèle adjoint car le temps de calcul est plus important. L'efficacité peut se calculer comme le rapport du temps de calcul sur le temps total d'exécution du programme parallèle. Lorsque l'on passe du modèle direct à son adjoint, on augmente la quantité de calculs sans rien changer d'autre, l'efficacité augmente donc.

En ce qui concerne le recouvrement des communications par le calcul, sur le Cray, on sait que la vitesse de calcul du processeur est faible par rapport au débit du réseau de communication. La durée des communications est donc proportionnellement très faible (moins de 10 ms sur 140 ms avec 16 processeurs) même sans recouvrement, il n'y a rien à gagner par cette technique.

Sur l'IBM au contraire, les processeurs sont très rapides par rapport au réseau. Le temps d'exécution du modèle direct sur 16 processeurs est d'environ 760 ms, c'est-à-dire 19 ms par pas de temps. On peut mesurer plus précisément les temps que prennent les différentes phases, on obtient les valeurs suivantes :

- 0.835 ms de calcul (pour un pas de temps),
- 0.146 ms pour l'empaquetage d'un message,
- 0.507 ms pour l'envoi d'un message,
- 0.081 ms pour le dépaquetage d'un message,
- 15.2 ms d'attente des messages par pas de temps.

Que se passe-t-il lorsqu'on recouvre les communications par le calcul ? La seule partie que l'on peut recouvrir est le temps d'attente des messages, les temps d'empaquetage, de dépaquetage et d'envoi des messages ne peuvent pas être recouverts. Pour faire ce

recouvrement, on peut utiliser le temps de calcul sur l'intérieur d'un domaine, c'est-à-dire 0.730 ms. On gagne donc 0.730 ms par pas de temps soit 29 ms au total. Le temps d'exécution du modèle direct avec recouvrement des communications est de 0.730 ms, tout se passe donc bien comme prévu mais la quantité de calcul disponible pour le recouvrement est trop faible pour que l'amélioration soit significative. Pour le modèle adjoint, il y a un peu plus d'opérations à effectuer, le résultat est très légèrement meilleur. On peut retenir de ces essais que pour espérer obtenir un masquage efficace, il faut au minimum que le calcul dure aussi longtemps que les communications. Cela signifie que pour un modèle peu coûteux, le nombre optimal de processeurs est faible.

5.2.3 Assimilation

Dans ce paragraphe, nous mettons en application les modèles parallèles que nous venons de tester dans un algorithme d'assimilation de données. Nous utilisons pour cet essai un algorithme de Newton tronqué. Pour cet essai, nous faisons une assimilation sur 40 pas de temps de 90 secondes, soit 1 heure, et une discrétisation spatiale de 122×122 points. La dimension de la variable de contrôle est alors 100800. Pour 10 itérations de cet algorithme et 10 pas de gradient conjugué à chaque étape, nous obtenons les performances du tableau 5.2.

Nb de Processeurs	1	4
Temps total (sec.)	392.8	369.9
Temps CPU (sec.)	365.6	322.6

TAB. 5.2 - *Assimilation de données avec les modèles parallèles sur IBM SP1*

Ceci nous donne donc une efficacité très faible (27 %). Dans les mêmes conditions, l'efficacité respective des modèles directs et adjoint est de 42% et 63%. Nous perdons de l'efficacité car l'algorithme de gradient conjugué nécessite le calcul de produits scalaires qui est une opération trop coûteuse en communications et qui conduit à ce mauvais résultat.

5.2.4 Conclusion

Les résultats présentés ici sont relativement mauvais. Il faut cependant les nuancer un peu car la même méthode appliquée à un modèle plus complexe donc plus coûteux en calcul donnerait de meilleurs résultats. En effet, sur le SP1 avec 16 processeurs, le temps d'exécution total (760 ms) se décompose en 34 ms de calcul, 36 ms pour l'empaquetage et le dépaquetage des messages, 81 ms pour l'envoi des messages et 610 ms d'attente. Nous retrouvons donc bien l'efficacité de la courbe 5.2 :

$$\frac{34}{727+34} = 5\%.$$

Imaginons maintenant un modèle plus sophistiqué qui demande non pas 62 mais 1200 opérations par point de grille (nous sommes encore très en dessous d'u modèle

réel comme ARPS qui demande environ 5850 opérations par points dans la version 4.0). Nous aurons alors un temps de calcul d'environ 650 ms et un temps total de ms. L'efficacité devient :

$$\frac{650}{727+650} = 47\%.$$

L'efficacité est donc augmentée de façon significative. Mais, on peut également utiliser avec profit la méthode de recouvrement des communications. Dans ce cas, les 610 ms d'attente seront entièrement masquées par les 650 ms de calcul. Le temps total sera donc de 767 ms et on atteindra

$$\frac{650}{767} = 85\%$$

d'efficacité.

De même, le temps d'exécution des produits scalaires dans l'algorithme du gradient conjugué deviendra faible par rapport au temps total et la parallélisation de l'assimilation de données sera performante.

Néanmoins, ces expériences confirment qu'un bon algorithme séquentiel peut ne pas être adapté à une implémentation parallèle. Il faut donc rechercher des algorithmes mieux adaptés aux architectures parallèles.

5.3 Couplage de sous domaines

Cette partie est consacrée à l'étude des performances de la parallélisation par couplage des sous-domaines tel que nous l'avons présenté au chapitre 4. Nous découpons le domaine de calcul de la même manière que précédemment mais l'algorithme d'assimilation est exécuté sur chaque sous-domaine avec un terme de pénalisation pour forcer la régularité de la solution entre les domaines.

5.3.1 Exemple de résultat numérique

Nous présentons ici un résultat pour une discrétisation de 21×31 points qui permet de visualiser le type de résultat que l'on peut obtenir avec cet algorithme. La période d'assimilation étant de 30 pas de 120 secondes, c'est-à-dire une heure. La condition initiale perturbée avant assimilation est représentée sur la figure 5.3. La figure 5.4 montre le résultat obtenu sur 6 sous-domaines de taille 11×11 avec un point de recouvrement aux interfaces. On constate que les limites des sous-domaines ne sont pas perceptibles, ce qui donne un aperçu de la validité de l'algorithme parallèle.

5.3.2 Problème à taille constante

Le test qui suit a été réalisé sur une période d'assimilation de 30 pas de 90 secondes avec un domaine global de taille 121×121. La dimension de la variable

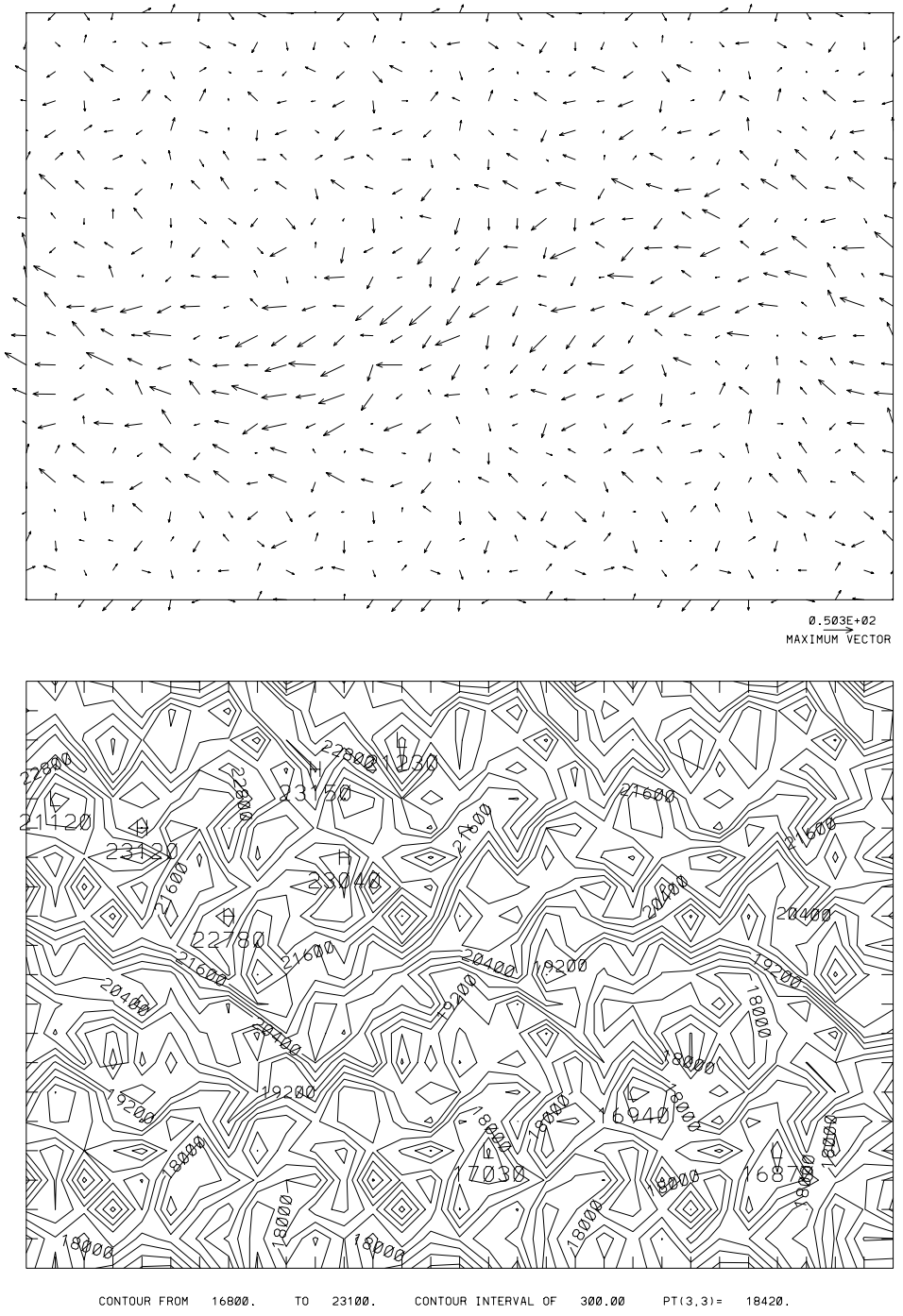


FIG. 5.3 - *Condition initiale perturbée.*

de contrôle est alors de 85323. Nous présentons ici les résultats obtenus sur IBM SP1.

Dans le tableau 5.3, la colonne *Pas de Newton* indique le nombre d'itérations de la méthode de Newton tronquée qui ont été nécessaires à la convergence, la co-

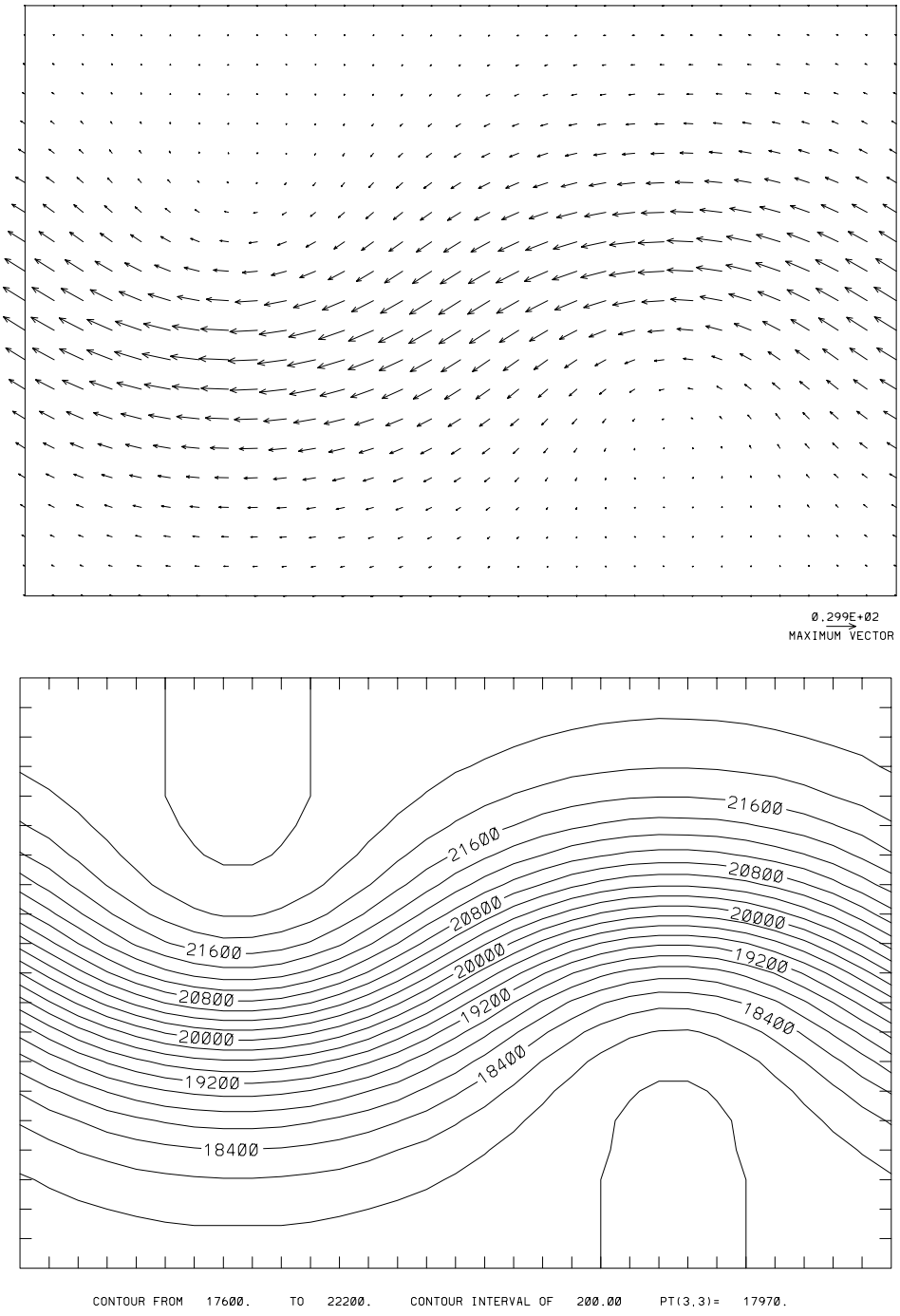


FIG. 5.4 - *Condition initiale après optimisation.*

lonne *Gradient* indique le nombre total de pas de gradient conjugué qui correspond au nombre d'intégrations du système adjoint au second ordre. Les deux colonnes suivantes indiquent les erreurs maximales sur le vent et le géopotentiel après minimisation. Le temps écoulé est le temps total d'exécution de l'algorithme, le temps

Nb. de Processeurs	Pas de Newton	Gradient (SOA)	Erreur max.		Temps (sec.)		Util. (%)	Eff. (%)
			u^2+v^2	Φ	écoulé	CPU		
1	5	28	7.9E-2	1.5E+1	80.8	77.7	-	-
1x2=2	5	60	4.8E-1	6.9E+1	83.8	83.0	97.0	48.2
2x1=2	5	42	4.3E-1	6.1E+1	62.1	61.5	98.7	65.1
4	5	75	5.6E-1	7.5E+1	63.9	52.3	79.3	31.6
3x2=6	5	77	4.6E-1	8.6E+1	38.1	37.5	90.9	35.3
2x3=6	5	81	7.5E-1	8.5E+1	46.0	37.8	75.3	29.3
2x4=8	6	121	5.0E-1	6.5E+1	49.7	41.5	70.4	20.3
4x2=8	5	77	4.3E-1	7.2E+1	28.8	28.2	89.7	35.1
9	5	82	5.4E-1	8.8E+1	26.8	25.9	88.7	33.5
4x3=12	5	99	4.8E-1	7.3E+1	44.6	22.5	40.3	15.1
3x4=12	5	96	5.7E-1	9.4E+1	37.0	23.4	49.4	18.2
16	5	84	5.3E-1	7.3E+1	29.8	15.3	45.1	16.9
4x5=20	5	91	5.3E-1	7.4E+1	25.9	13.2	43.2	15.6
25	6	120	4.3E-1	6.6E+1	27.1	15.0	43.1	11.9
5x6=30	6	123	3.3E-1	5.4E+1	27.6	11.8	34.9	9.8
6x5=30	5	86	4.8E-1	6.6E+1	19.6	8.8	38.9	13.7

TAB. 5.3 - *Résultats du problème à taille constante.*

CPU est le maximum des temps d'exécution sur les différents processeurs. Les deux dernières colonnes donnent les pourcentages d'utilisation des processeurs et l'efficacité du programme parallèle. Le pourcentage d'utilisation représente le pourcentage de temps pendant lesquels les processeurs ont été actifs pendant le déroulement du programme (les phases d'initialisation et d'entrées/sorties ne sont pas prises en compte). Sa valeur est donnée par :

$$U = \frac{\sum_{N_{proc}} T_{CPU}}{N_{proc} * T_{Ecoule}}.$$

On constate sur la figure 5.5 que ce pourcentage devient très faible au delà de 10 processeurs. Cela peut s'expliquer de deux façons : soit le temps de communication est trop important par rapport au temps de calcul, soit le calcul est trop déséquilibré entre les processeurs. Comparons les figures 5.6 et 5.7 qui représentent les diagrammes de Gantt des exécutions avec 9 et 16 processeurs, chaque ligne représente le temps d'activité (en noir) et d'inactivité (en blanc) d'un processeur. Le pourcentage d'utilisation donné dans le tableau 5.3 est équivalent au pourcentage de surface noir dans le diagramme. L'utilisation est bonne avec 9 processeurs (88.7%), on vérifie sur la figure 5.6 que tous les processeurs sont actifs la majeure partie du temps. L'intervalle de temps entre la fin d'une étape de minimisation et le début de la suivante est très faible : le temps de communication n'est donc pas le problème. Avec 16 processeurs, l'utilisation est beaucoup plus faible (45.1%), ce qui est confirmé par la figure 5.7. L'information supplémentaire que nous apporte ce diagramme est que seulement 3 processeurs restent actifs une grande partie du temps. Les autres sont bloqués en attente des valeurs aux interfaces : le calcul est très mal équilibré.

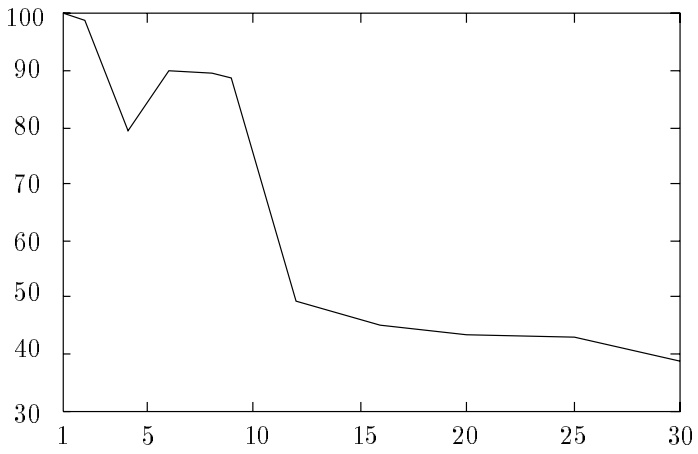


FIG. 5.5 - *Utilisation des processeurs*

Un autre facteur peut influencer l'équilibrage de la charge. Avec 8 processeurs par exemple, on peut décomposer le domaine de deux manières différentes (4 domaines dans la direction est-ouest, 2 dans la direction nord-sud ou bien le contraire). On constate dans le tableau 5.3 que le nombre d'itérations et *a fortiori* le temps de calcul n'est pas le même dans les deux cas. Les données et la manière de répartir les domaines ont une grande influence sur l'équilibrage de la charge.

Malheureusement, ce problème n'est pas facile à résoudre. La première idée qui vient à l'esprit est de diminuer la taille des domaines où la convergence est la plus lente pour augmenter les autres. Mais la vitesse de convergence peut changer complètement si on modifie les domaines comme cela a été montré avec 8 processeurs. Une autre solution serait de définir plus de sous-domaines que nous avons de processeurs. La charge pourrait alors être gérée en déplaçant les sous-domaines d'un processeur à un autre dynamiquement. En appliquant cela, nous devons toutefois prendre garde à ne pas trop augmenter la charge totale de calcul due au traitement des interfaces, utiliser des sous-domaines trop petits sera certainement pénalisant par cet aspect.

Dans les deux cas, une difficulté supplémentaire vient du fait que la vitesse de convergence sur un même sous-domaine peut varier d'une itération à l'autre de la méthode de Newton, même sans changer la définition des sous-domaines. On peut constater cela sur la figure 5.7. Le temps de calcul pour le processeur 7 entre le quatrième et le cinquième pas de Newton est augmenté d'un facteur environ 6. Sur ce sous-domaine, la convergence était la plus rapide à la quatrième itération et devient l'une des plus lentes à l'itération suivante alors qu'aucune modification n'est intervenue.

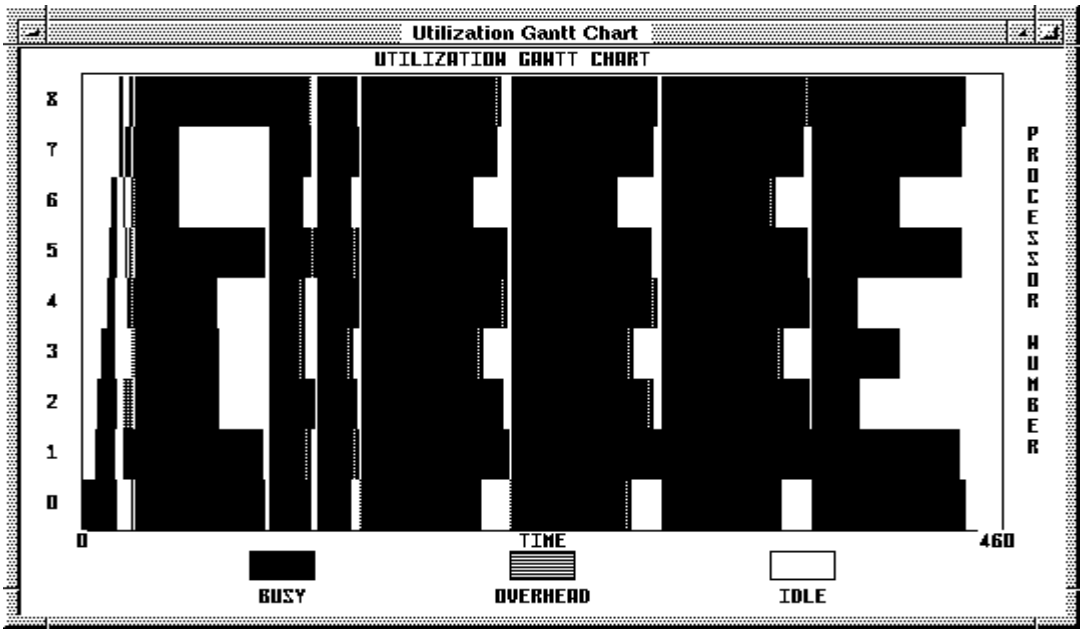


FIG. 5.6 - *Diagramme de Gantt avec 9 processeurs*

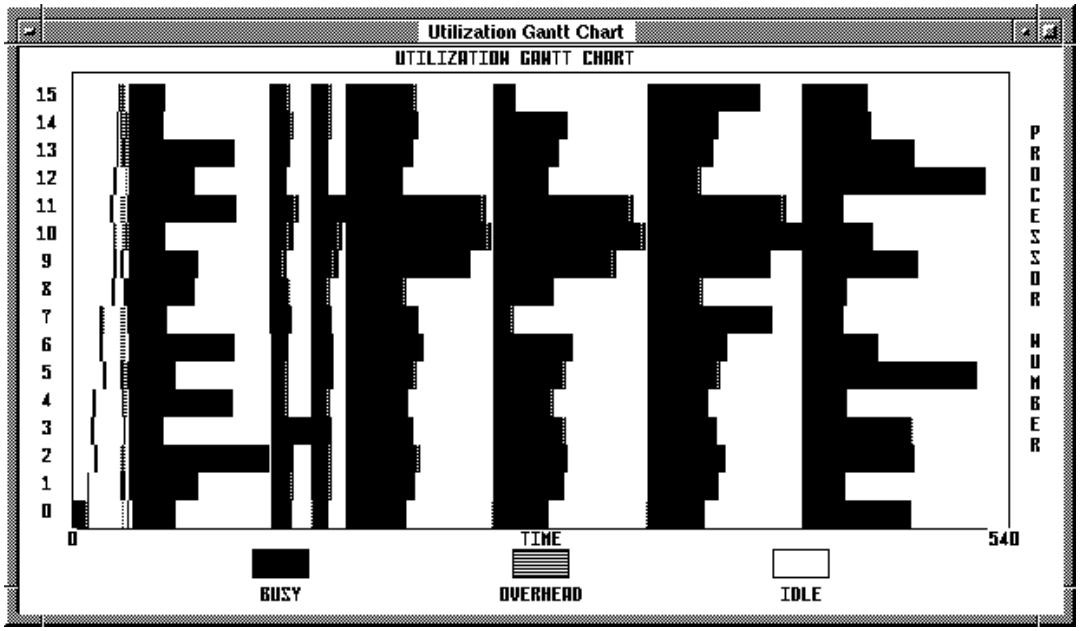


FIG. 5.7 - *Diagramme de Gantt avec 16 processeurs*

Tous ces facteurs expliquent l'efficacité relativement médiocre que l'on atteint avec cet algorithme. Nous testons dans le paragraphe suivant le même algorithme en distribuant 2 sous-domaines par processeur avec une numérotation rouge-noir.

5.3.3 Algorithme rouge-noir

Dans cette partie nous proposons une méthode pour améliorer l'équilibrage de charge de notre algorithme parallèle. Elle consiste à colorier les sous-domaines avec un coloriage de type «rouge-noir» et à affecter un domaine rouge et un domaine noir à chaque processeur. La différence avec l'essai précédent est que nous avons une phase de communication chaque fois qu'un pas de minimisation a été réalisé sur la moitié des sous-domaines. Pour une itération donnée, la seconde moitié des sous-domaines commence une étape de minimisation avec plus d'information que dans la version précédente de l'algorithme. Les résultats sur le SP1 sont donnés ci-dessous :

Nb. de Proc.	Nb. de Domaines	Pas de Newton	Gradient (SOA)	Temps (sec.)		Util.	Eff.
				écoulé	CPU	(%)	(%)
1	1	5	28	80.8	77.7	-	-
2	4	5	78	105.3	54.2	92.4	38.4
4	2x4=8	6	93	67.6	32.9	80.5	29.9
4	4x2=8	5	81	60.5	29.7	82.2	33.4
5	9	6	103	67.6	32.2	70.5	23.9
8	16	6	95	44.2	17.3	67.0	22.9

Résultats du problème à taille constante
par la méthode rouge-noir.

Le premier résultat que l'on constate dans ce tableau est que l'efficacité reste du même ordre de grandeur que dans le paragraphe précédent. Le fait de grouper deux sous-domaines sur un seul processeur ne change rien au problème de l'équilibrage de charge (cela se vérifie sur la figure 5.8). Tant que nous n'aurons aucun moyen de prévoir la charge relative des différents sous-domaines, nous ne saurons pas les répartir de manière équitable.

Le second résultat que l'on obtient est que le nombre d'itérations pour un sous-domaine donné peut être plus important qu'il ne l'était avec la première méthode. Plus précisément, on peut voir sur les figures 5.9 et 5.10 que le nombre d'itérations augmente pour les sous-domaines qui se trouvent dans le groupe des sous-domaines par lesquels on commence le calcul et décroît pour les autres.

5.3.4 Problème à charge constante

Ce paragraphe est consacré aux résultats obtenus en augmentant la taille du problème (c'est-à-dire de la grille de discrétisation) proportionnellement au nombre de processeurs. On garde ainsi une charge constante sur les processeurs (en terme de coût mémoire et de temps CPU nécessaire à l'intégration des systèmes directs et adjoints). Ces essais ont été réalisés sur une période de 6 heures (120 pas de 180 secondes). La taille du problème de minimisation sur chaque processeur est gardée constante, la dimension de la variable de contrôle étant 28443. Les tableaux 5.4 et 5.5 donnent les résultats obtenus pour différents essais sur IBM SP1 et Cray T3D.

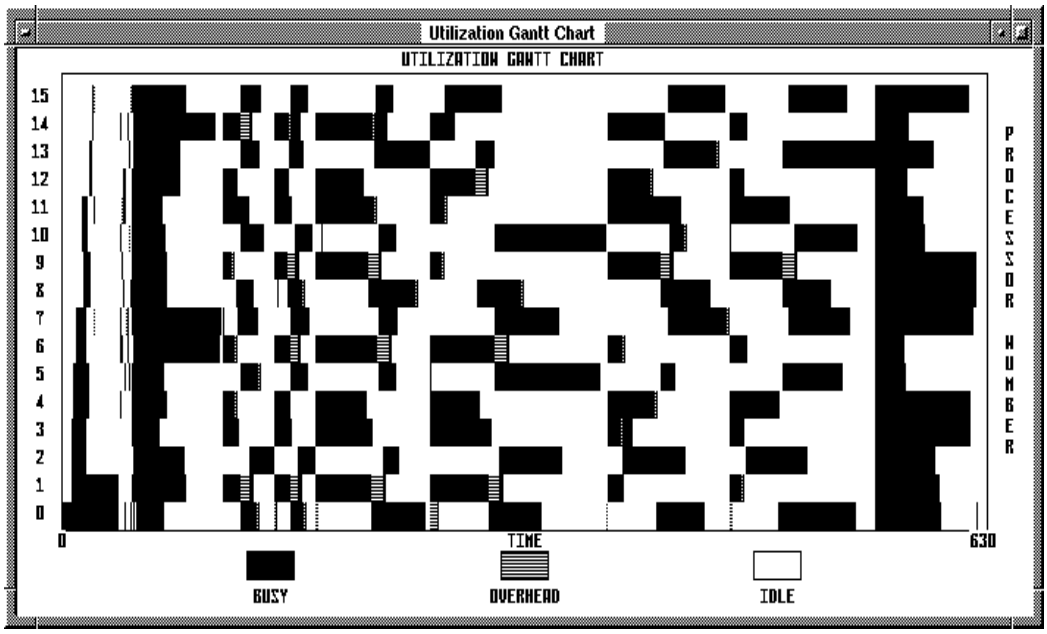


FIG. 5.8 - *Diagramme de Gantt pour 16 sous-domaines sur 8 processeurs*

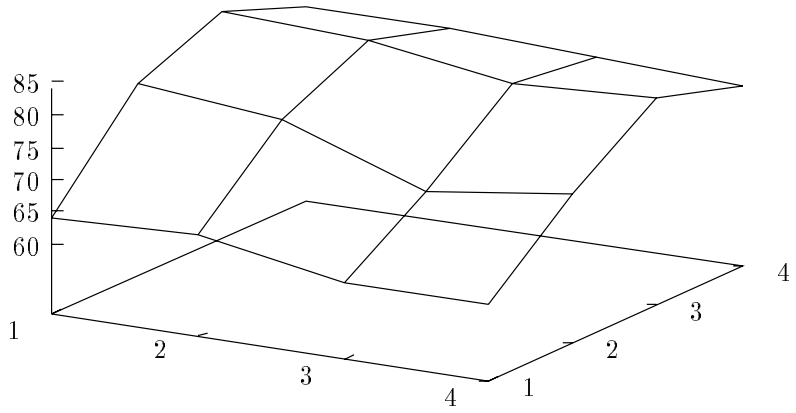


FIG. 5.9 - *Nombre d'itérations par sous-domaine.*

Nombre de Processeurs	Pas de Newton	Gradient (SOA)	Erreur max.	
			u^2+v^2	Φ
1	4	27	1.1E-2	2.0E0
4	5	31	1.3E-2	2.1E0
9	5	42	2.0E-2	3.9E0
16	5	45	4.3E-2	6.8E0

TAB. 5.4 - *Résultats de convergence.*

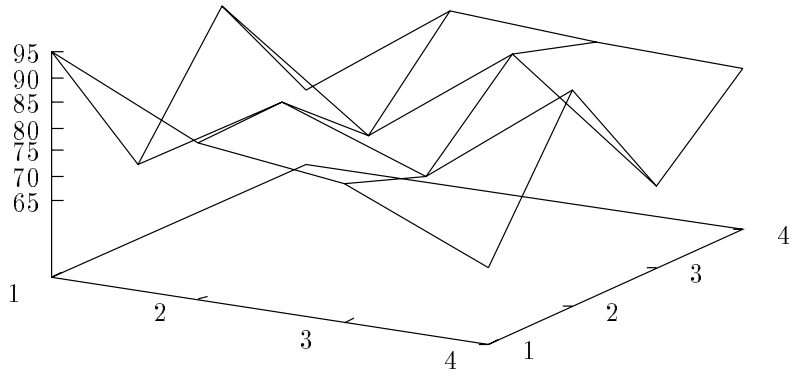


FIG. 5.10 - *Nombre d'itérations par sous-domaine en utilisant la méthode rouge-noir.*

Nombre de Processeurs	IBM SP1		Cray T3D	
	écoulé	CPU	écoulé	CPU
1	8.8	8.7	19.7	18.7
4	12.6	12.1	23.6	22.5
9	16.7	15.3	-	-
16	17.9	16.9	33.1	31.5

TAB. 5.5 - *Temps de calcul.*

On constate sur le premier tableau que la précision obtenue est du même ordre de grandeur lorsque le nombre de processeurs augmente. Le nombre d'itérations avant convergence augmente faiblement, cette augmentation n'est pas très importante par rapport à l'augmentation de la taille du problème global traité. Ce tableau montre donc la validité de l'algorithme. Le second tableau montre les temps de calcul obtenus sur l'IBM SP1 et le Cray T3D. Les temps d'exécution sont proportionnels au nombre d'itérations réalisées sur les deux machines. Nous remarquons que le SP1 est sensiblement plus rapide que le T3D, cela est dû aux vitesses relatives atteintes sur les processeurs de ces deux machines. Les meilleures performances du T3D en communication n'apportent pas d'amélioration ici car elles ne sont pas le facteur déterminant.

Les résultats de ce paragraphe semblent meilleurs que ceux obtenus dans les paragraphes précédents. Cela est en partie dû au fait que la période d'assimilation n'est pas la même et que cela modifie la vitesse de convergence de l'algorithme. L'équilibrage de charge est meilleur, même avec 16 processeurs. Un autre bon résultat est que l'algorithme parallèle nous permet de traiter des problèmes d'une taille que nous ne pouvons pas atteindre avec le programme séquentiel à cause des contraintes de mémoire.

5.3.5 Conclusion

L'algorithme parallèle que nous présentons ici est mal équilibré en terme de charge de calcul. Il nous faudrait utiliser des mécanismes de régulation dynamique de la charge pour les améliorer. Mais, dans le problème que nous considérons, nous n'avons pas de critère permettant d'évaluer *a priori* le coût d'une itération sur un sous-domaine. Ce coût dépend fortement des données initiales, du résultat de l'itération précédente et des valeurs aux interfaces provenant des sous-domaines voisins.

Le dernier essai nous montre cependant que cette méthode peut être utilisée avec de bonnes performances pour résoudre certains problèmes. La difficulté réside dans la détermination des problèmes pour lesquels cet algorithme sera efficace, les nombreux facteurs qui interviennent n'étant pas tous connus avant l'exécution.

5.4 Décomposition en temps

Dans cette partie, nous faisons également une décomposition du domaine de calcul mais cette fois dans la direction du temps. Nous couplons des modèles sur le même domaine géographique mais sur des périodes de temps différentes, l'interface est donc constituée des conditions initiales et finales. Nous couplons alors les sous-domaines par le même algorithme que dans la partie précédente : l'échange des conditions aux limites est remplacé par un échange des conditions initiales et finales entre deux sous-domaines successifs.

Nous présentons ici les résultats obtenus pour deux variantes de l'algorithme : une version synchrone dans laquelle une phase d'échange des conditions initiales et finales a lieu entre chaque pas de la méthode de Newton et une version asynchrone dans laquelle, à la fin de chaque pas, un processeur donné prend en compte les messages qui sont arrivés pendant le calcul du pas précédent et effectue le pas suivant sans nécessairement attendre les messages du processeur qui le précède ou le suit. Il se peut aussi que l'un ou les deux processeurs adjacents aient envoyé plusieurs messages car ils ont effectué plusieurs itérations, tous les messages sont alors pris en compte. Cette variante permet *a priori* de minimiser les temps d'attente et donc d'augmenter le temps d'activité « utile » des processeurs.

5.4.1 Performances

Nous nous intéressons dans un premier temps à l'assimilation de données sur une période de 8 heures pour un domaine discrétisé en 21×21 points. Nous décomposons ce domaine dans la direction du temps, chaque processeur aura donc la charge du domaine complet sur une période de temps plus courte. Les résultats de ces essais sont présentés sur les figures 5.11, 5.12 et 5.13.

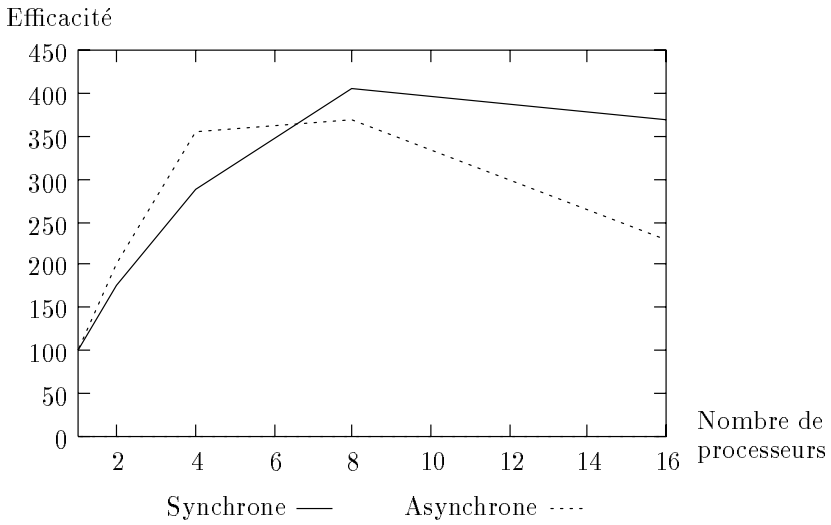


FIG. 5.11 - *Efficacité de la décomposition de domaine dans la direction du temps.*

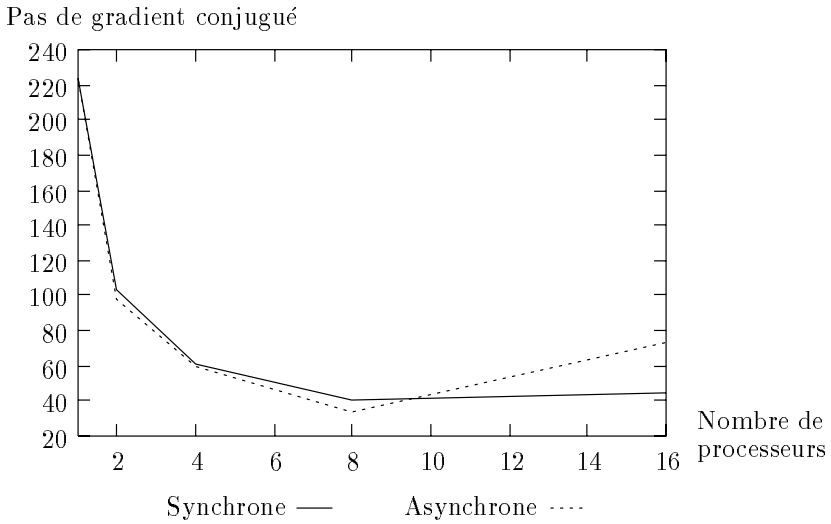


FIG. 5.12 - *Nombre maximum de pas de gradient conjugué.*

Le premier résultat que l'on constate (voir figure 5.11) est que l'efficacité de cette méthode reste supérieure à 100 % de 2 à 16 processeurs pour les deux implémentations de l'algorithme. Pourtant, la taille du problème a été choisie pour que même avec un seul processeur il n'y ait pas de problème de mémoire (et donc de temps de `swap` inutile). L'algorithme utilisé est donc meilleur que l'algorithme de référence dans ce cas.

La figure 5.12 montre le nombre maximum d'itérations de gradient conjugué réalisées par processeur au cours de l'assimilation. On pouvait s'attendre à ce qu'une

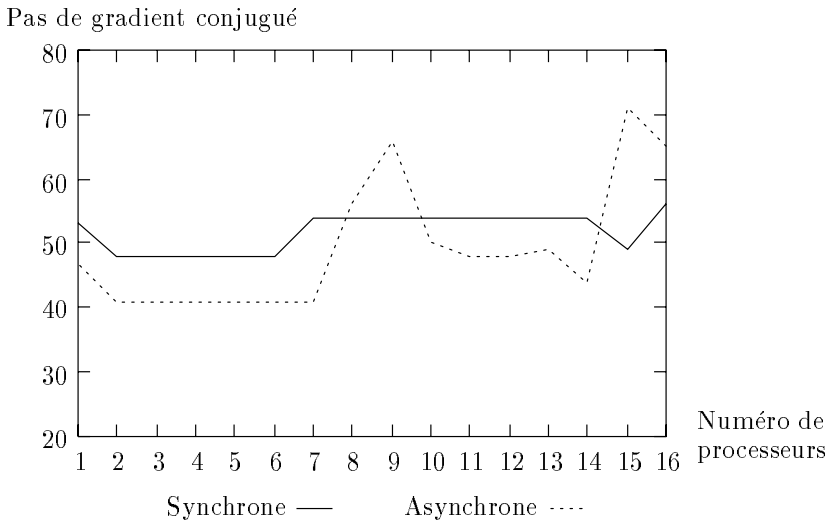


FIG. 5.13 - *Nombre de pas de gradient conjugué par processeur.*

itération dure moins longtemps en parallèle mais que le nombre d'itérations varie peu, or on constate que ce nombre diminue fortement. Avec un seul processeur, on doit résoudre un problème d'optimisation de taille 74271, ce qui nécessite 20 pas de Newton et 224 pas de gradient conjugué au total. Avec 8 processeurs par exemple, on doit résoudre 8 problèmes d'optimisation de taille 10431, ce qui demande 9 pas de Newton et 40 itérations de gradient conjugué au maximum. On a donc 5.5 fois moins d'itérations et une itération coûte 7 fois moins d'où une accélération théorique de 38 (32 en réalité à cause du coût des communications et des tests du critère d'arrêt).

L'ajout à la fonction de coût d'un terme de continuité entre les sous-domaines qui varie d'un pas à l'autre devrait perturber la convergence de l'algorithme de minimisation. On constate que cette perturbation est suffisamment faible pour conserver l'avantage que procure la réduction de la taille du problème.

La figure 5.13 nous montre, pour 16 processeurs, la distribution du nombre d'itérations de gradient conjugué sur les processeurs. On constate que la version asynchrone est plus déséquilibrée, c'est ce qui explique sa moins bonne efficacité. On peut aussi remarquer que plus le nombre de processeurs augmente et plus la méthode asynchrone perd de son efficacité par rapport à la méthode synchrone. On notera aussi que plus on utilise de processeurs et plus le comportement de la méthode asynchrone est chaotique d'une exécution à l'autre. On peut trouver dans le tableau 5.6 des exemples de résultats obtenus sur 16 processeurs pour résoudre le même problème. Le cas qui se produit le plus souvent (plus de 90 % des cas) est le premier dans le tableau, malheureusement, c'est aussi le moins bon. Sur le SP1, ce comportement chaotique est renforcé par les perturbations de la machine dues au système d'exploitation.

Temps (secondes)	Newton	Gradient			Accélération
		Min	Max	Moy	
5.5	10	39	73	45	36.6
4.3	10	32	45	41	46.8
5.2	9	31	66	41	38.7
2.3	4	20	28	27	87.4
4.8	8	25	60	40	41.9
4.5	10	32	45	40	44.7
5.1	10	32	66	44	39.4
5.2	9	30	57	38	38.7
4.8	9	32	62	41	41.9

TAB. 5.6 - *Exemples de performances sur 16 processeurs.*

5.4.2 Extensibilité

Dans ce paragraphe, nous nous intéressons à une assimilation de données sur un domaine de taille 41×41 et nous gardons une période d'une heure par processeur, c'est-à-dire que la période d'assimilation augmente avec le nombre de processeurs.

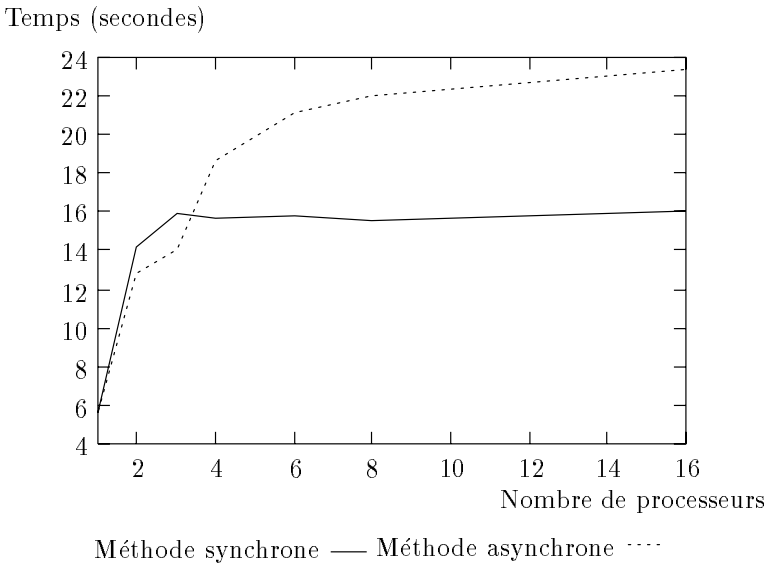


FIG. 5.14 - *Extensibilité de la décomposition en temps*

Les figures 5.14 et 5.15 représentent, toujours pour les deux variantes de l'algorithme, le temps et le nombre d'itérations de gradient conjugué en fonction du nombre de processeurs. On constate que le temps d'exécution est proportionnel à ce nombre d'itérations sauf avec deux processeurs. On remarque également que la méthode synchrone se montre la plus performante à partir de 4 processeurs. Enfin,

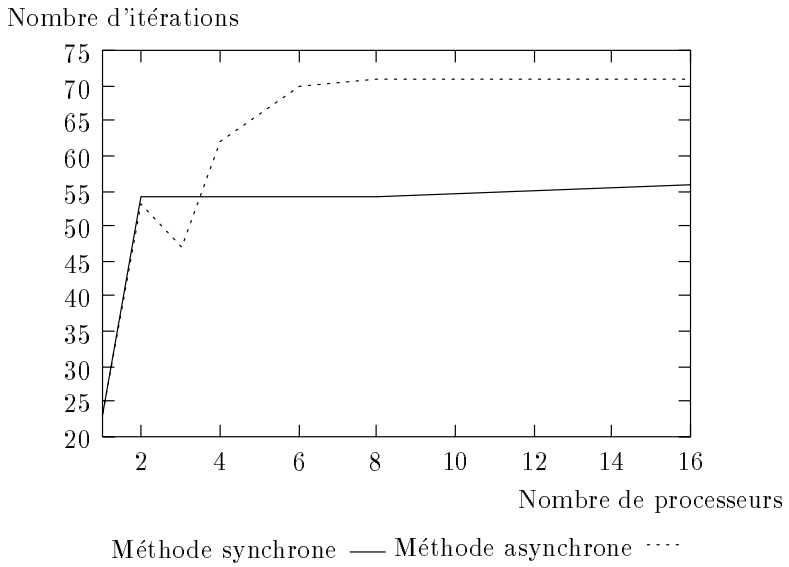


FIG. 5.15 - *Extensibilité de la décomposition en temps*

on remarque que le temps d'exécution reste constant à partir de trois processeurs pour la méthode synchrone: avec un processeur il n'y a aucune interface à prendre en compte, avec deux processeurs, chacun doit prendre en compte une interface et à partir de trois processeurs, chacun d'eux a au plus deux interfaces à prendre en compte. Ceci montre la très bonne extensibilité de cet algorithme. Cela signifie que l'on peut augmenter la durée de la période d'assimilation et que si l'on augmente proportionnellement le nombre de processeurs, on aura un temps de réponse constant.

5.4.3 Conclusion

Les résultats présentés dans cette partie sont meilleurs que ceux obtenus en décomposant le domaine de calcul en espace, mais, il ne faut pas généraliser ce résultat. En effet, on peut rencontrer le même phénomène pour une décomposition domaine en espace lorsque la période d'assimilation est très courte : la dimension de la variable interface sera alors petite devant la dimension de la variable de contrôle décomposée. Cette explication reste à confirmer dans un plus grand nombre de cas et surtout avec des modèles plus sophistiqués que celui utilisé ici.

Cet exemple montre également que même pour une exécution parallèle, un algorithme synchrone est parfois préférable à sa version asynchrone : il est inutile d'itérer trop une minimisation locale sans informations globales supplémentaires.

5.5 Conclusion

Nous avons montré ici que la parallélisation ouvre également de nouvelles perspectives dans le domaine du couplage de modèles. Ce point devient de plus en plus important dans l'amélioration souhaitée de la précision des prévisions. L'algorithme proposé est assez efficace pour être appliqué avec succès à une certaine classe de problèmes. Néanmoins, il sera nécessaire de l'adapter à chaque cas particulier en fonction de la configuration des modèles et des échelles de temps et d'espace utilisés.

Afin d'améliorer les résultats donnés ici, il serait nécessaire d'étudier une version asynchrone de l'algorithme incluant un mécanisme de régulation dynamique de la charge. Ce problème est compliqué par le caractère imprévisible de la charge de calcul d'une itération à l'autre.

Enfin, une extension importante reste à valider dans le cas du couplage de modèles différents.

Chapitre 6

Un modèle réel : ARPS

Dans ce chapitre, nous présentons le modèle météorologique ARPS (Advanced Regional Prediction System) qui est un modèle régional conçu pour la prévision des tornades. Nous présenterons une parallélisation de ce modèle écrite en collaboration avec K. Johnson (SCRI, Florida State University) et présenterons les performances obtenues.

6.1 Introduction

6.1.1 Présentation du projet ARPS

ARPS (Advanced Regional Prediction System) est un modèle régional développé spécialement pour la prévision des tornades par le CAPS (Center for Analysis and Prediction of Storms) à l'Université de l'Oklahoma. Ce projet fait partie d'un groupe de onze projets moteurs pour le développement de la science aux États-Unis lancés en 1988 par la NSF (National Science Foundation).

Ce code a été écrit pour être utilisé avec un réseau de 175 radars Doppler (réseau NEXRAD) qui couvre le territoire des États-Unis. Le modèle ARPS est un prototype du modèle régional que devra intégrer chaque centre en fonction des données provenant de son propre radar et éventuellement des radars voisins. Les phénomènes météorologiques considérés s'étendent sur une échelle qui va de quelques kilomètres pendant quelques dizaines de minutes (orages) jusqu'à quelques centaines de kilomètres pendant plusieurs heures (systèmes dépressionnaires et systèmes méso-échelle).

Ces phénomènes ne sont pas modélisables par les modèles classiques utilisés en météorologie opérationnelle (prévision à moyen terme). Les phénomènes physiques de petite échelle sont plus complexes, notamment en ce qui concerne les phénomènes convectifs, le cycle de l'eau et de façon plus générale la micro-physique. Donc la modélisation numérique s'appuiera sur des modèles non-hydrostatiques et compressibles pour simuler certains processus physiques. Les solutions de ces modèles contiennent des ondes sonores ce qui diminue très significativement le pas d'inté-

gration temporelle pour assurer la stabilité du schéma d'intégration. Cela implique une augmentation importante de la puissance de calcul nécessaire pour intégrer ces modèles. De plus, les observations ne sont pas régulièrement fournies comme elles le sont par le réseau synoptique de la prévision à moyen terme. C'est pourquoi il se met actuellement en place un réseau de radars sur le territoire des États-Unis.

On trouve dans [14] et [15] les principales caractéristiques du code qui sont :

- Modèle tridimensionnel, non-hydrostatique et compressible.
- Système de coordonnées curvilignes suivant le terrain.
- Code modulaire pour faciliter les améliorations progressives.
- Documentation interne et externe importante.
- Utilisation d'opérateurs différentiels pour améliorer la lisibilité du code.
- Code écrit en Fortran 77 « pur » pour avoir la plus grande portabilité possible.
- Possibilité de maillage adaptatif.

6.1.2 Pourquoi paralléliser ?

La version ARPS 3.0 s'exécute à la vitesse de 165 Mflops sur un processeur Cray Y-MP pour un domaine de calcul de 64x64x32 points. Cela équivaut à environ 90000 mises-à-jour de points de grille par seconde. Un IBM RS6000 (modèle 530H) est environ 16 fois plus lent. Mais, dans la prévision météorologique, cela n'a pas grande signification. Ce qui nous importe est que le modèle s'exécute plus rapidement que l'évolution de l'atmosphère. On estime que pour que la prévision soit exploitable, le rapport du temps du modèle au temps physique réel doit être inférieur à un centième. Par exemple, une prévision à 4 heures doit s'exécuter en moins de 2 minutes 30. Pour une taille de grille opérationnelle (1000x1000x20), cela nécessite une vitesse de calcul utile de l'ordre du Tflops (10^{12} opérations flottantes par seconde). La figure 6.1 (d'après [14]) nous donne une idée de ce que l'on peut atteindre pour différentes vitesses de calcul.

On peut aussi évaluer la mémoire nécessaire à l'exécution d'un modèle opérationnel : il faut environ 12 Go de mémoire vive. Actuellement, on ne connaît pas de technologie qui permette d'atteindre une telle capacité de stockage sur une seule mémoire.

La seule solution raisonnable pour obtenir de telles performances à la fois de vitesse de calcul et de capacité mémoire est d'utiliser une machine parallèle à mémoire distribuée.

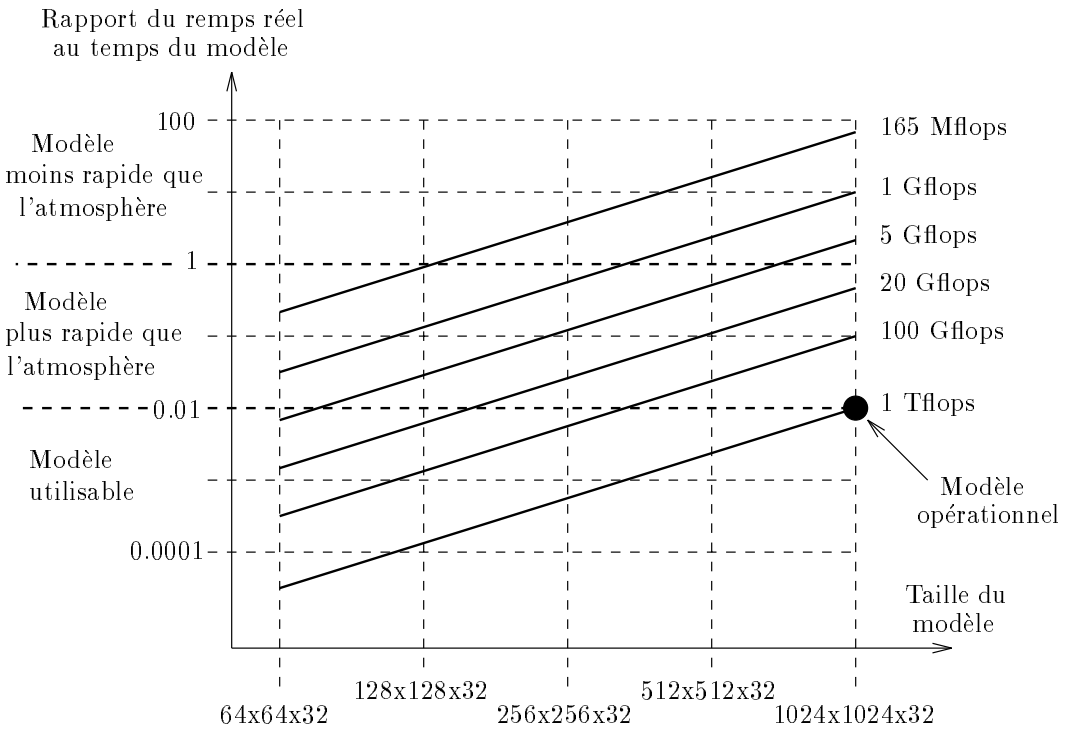


FIG. 6.1 - *Rapport entre le temps simulé et le temps réel pour différentes vitesses de calcul.*

6.2 Le modèle

6.2.1 Formulation

Les principales caractéristiques du modèle utilisé dans ARPS sont :

- Les équations de moment sont tri-dimensionnelles, non-hydrostatiques et compressibles.
- L'état de référence de l'atmosphère est hydrostatique, horizontalement homogène et indépendant du temps.
- Les termes de gradient de pression dans les équations de moments sont linéarisés autour de l'état de référence afin d'obtenir une plus grande précision numérique, on utilise comme variable l'écart de pression par rapport à cet état de référence.
- La variable thermodynamique est le potentiel de température.
- Pour prendre en compte le terrain, les équations ne sont pas écrites dans le système de coordonnées cartésiennes (x, y, z) mais dans un système de coordonnées curvilignes (ξ, η, ζ).

- On prend en compte les phénomènes d'échelle inférieure au pas de grille par un processus de turbulence.
- On ajoute un terme de dissipation artificiel pour atténuer les ondes sonores.
- La paramétrisation micro-physique tient compte de trois formes de l'eau : la vapeur d'eau, les nuages d'eau et l'eau de pluie.
- On peut utiliser des conditions aux limites rigides, périodiques, de gradient nul, d'ondes rayonnantes (radiating) ou des valeurs imposées.

Le modèle ARPS est composé des équations de conservation de la quantité de mouvement, de la masse (pression) et de l'eau ainsi que des équations d'état et de l'équation de la chaleur. Le potentiel de température et la pression sont déduites des équations de conservation d'énergie (thermique) et de la masse. Six valeurs correspondant aux états de l'eau (vapeur, nuages, eau de pluie, glace des nuages, neige et grêle) sont calculées. Leurs interactions et leur influence sur le thermique sont décrites par les équations de la micro-physique.

6.2.2 Résolution numérique

Discrétisation en temps

Nous avons vu que les équations utilisées dans ARPS prennent en compte la compressibilité de l'atmosphère. Cela implique l'apparition d'ondes sonores qui n'ont pas de signification météorologique dans les solutions calculées. Malheureusement, ces ondes réduisent considérablement le pas de temps utilisable dans un schéma explicite. Pour améliorer les performances du modèle, on doit donc utiliser la technique de *mode splitting*. Cette technique consiste à diviser un grand pas d'intégration en un certain nombre de petits pas de temps et à calculer les termes qui interviennent dans les ondes sonores à chaque petit pas de temps alors que les autres valeurs ne seront calculées qu'à chaque grand pas de temps. Les termes qui évoluent lors de chaque petit pas de temps sont les termes de gradient de pression dans les équations de moments et les termes de divergence dans l'équation de pression.

L'intégration en temps sur les grands pas de temps se fait selon un schéma saute-mouton. L'intégration sur les petits pas de temps utilise la méthode de prédiction-correction : on intègre les équations de moments sur un petit pas de temps pour déterminer le nouveau champ de vitesse puis on intègre dans le sens rétrograde l'équation de pression à partir de ces valeurs.

Discrétisation spatiale

Les équations discrètes du modèle ne sont pas écrites dans le système de coordonnées orthonormales (x, y, z) mais à l'aide de coordonnées curvilignes (ξ, η, ζ) mieux adaptées. Les valeurs des variables utilisées dans le modèle sont calculées sur

une grille « Arakawa C » dans le système de coordonnées précédent. Ce schéma consiste à calculer les valeurs vectorielles (composantes du vent) sur une grille et les valeurs scalaires sur une seconde grille dont les points sont décalés d'un demi pas de grille (voir figure 6.2). Cela permet notamment d'avoir des différences finies et des moyennes centrées et de prendre plus facilement en compte le terrain.

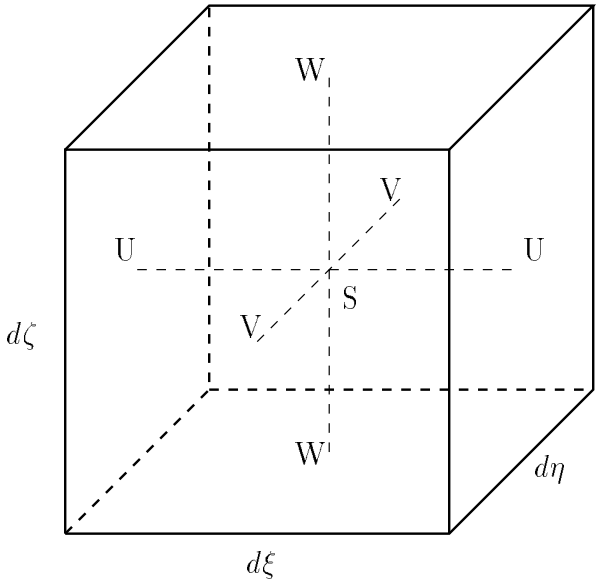


FIG. 6.2 - *Une cellule de la grille Arakawa C dans ARPS.*

Conditions aux limites

Pour un modèle régional tel que ARPS, seule la limite inférieure a une signification physique, sur les autres faces, les conditions aux limites sont artificielles. Le système de grille décrit plus haut pose des problèmes de définition des conditions aux limites. On doit définir des points extérieurs au domaine physique pour pouvoir calculer tous les termes des équations. La figure 6.3 nous montre cela dans le plan horizontal, le principe est le même dans toutes les directions.

La convention utilisée dans ARPS est que les dimensions horizontales des tableaux sont toujours `(1:nx,1:ny)`. Ils ne sont donc pas utilisés entièrement :

- `U` est défini pour `i=1` à `nx` et `j=1` à `ny`. Les conditions aux limites sont données à `i=1` et `nx` dans la direction ξ et `j=1` et `ny-1` dans la direction η. L'intégration est calculée pour `i=2` à `nx-1` et `j=2` à `ny-2`.

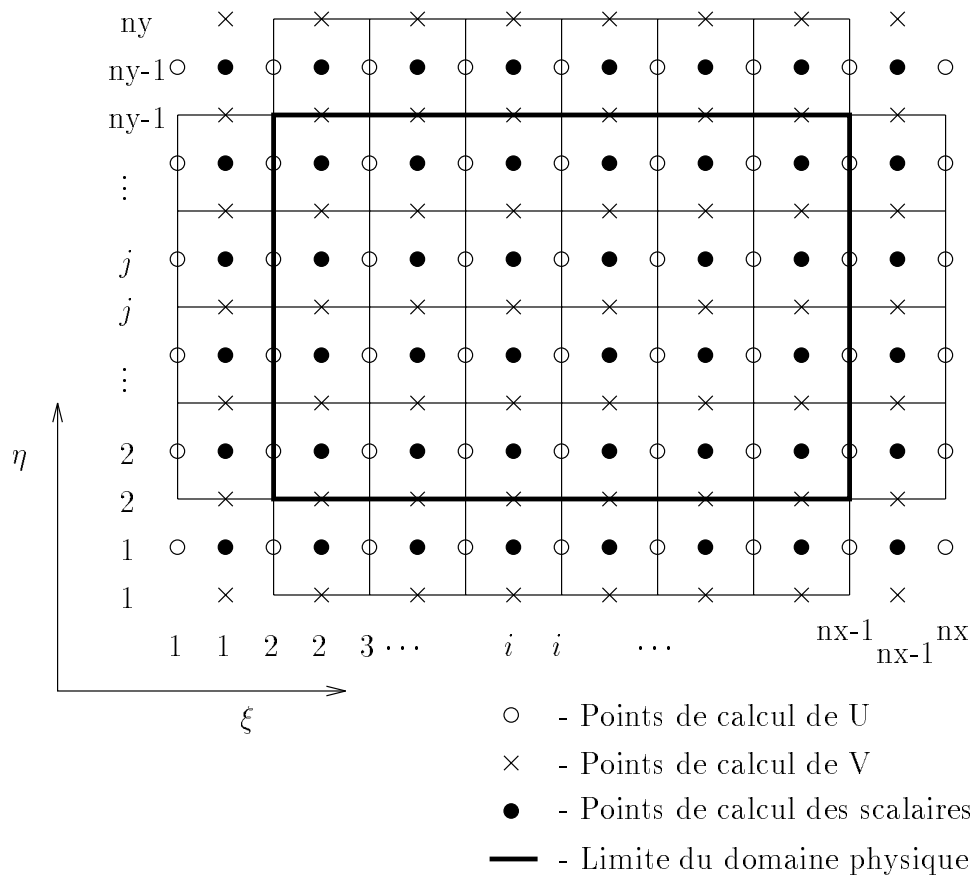


FIG. 6.3 - *Grille Arakawa C et conditions aux limites.*

- `V` est défini pour `i=1` à `nx` et `j=1` à `ny`. Les conditions aux limites sont données à `i=1` et `nx-1` dans la direction ξ et `j=1` et `ny` dans la direction η. L'intégration est calculée pour `i=2` à `nx-2` et `j=2` à `ny-1`.

- `W` est défini pour `i=1` à `nx-1` et `j=1` à `ny-1`. Les conditions aux limites sont données à `i=1` et `nx-1` dans la direction ξ et `j=1` et `ny-1` dans la direction η. L'intégration est calculée pour `i=2` à `nx-2` et `j=2` à `ny-2`.

- Les valeurs scalaires (pression, température, humidité) sont définies pour `i=1` à `nx-1` et `j=1` à `ny-1`. Les conditions aux limites sont données à `i=1` et `nx-1` dans la direction ξ et `j=1` et `ny-1` dans la direction η. L'intégration est calculée pour `i=2` à `nx-2` et `j=2` à `ny-2`.

On constate que toutes les variables ne sont pas définies et utilisées avec les mêmes bornes. Cela implique que les modifications à apporter lors de la parallélisation seront différentes.

6.2.3 Implémentation

Lorsqu'on découvre pour la première fois le code de ARPS, on remarque immédiatement qu'il a été écrit très « proprement ». Par exemple, chaque routine contient :

- une description des fonctionnalités,
- l'auteur, la date et l'historique des modifications,
- la liste des entrées et sorties et leur définition,
- l'usage systématique de `IMPLICIT NONE`,
- la liste des fichiers inclus,
- la liste des procédures et fonctions externes,
- le corps de la procédure.

Cela est indispensable pour que des non-spécialistes d'un code aussi volumineux puissent « entrer » dans le code, le comprendre, le modifier et donc le paralléliser. Cet aspect mérite d'être signalé car cela n'est pas vrai de tous les grands codes de simulation numérique.

Les opérateurs différentiels

La technologie des langages de programmation est telle que pour programmer efficacement, on est souvent obligé de s'éloigner de la structure mathématique du problème. On ne programme pas en termes de dérivées ou d'intégrales mais de boucles, de tests, etc... Cela a deux inconvénients : il est plus difficile de « rentrer » dans le programme et de le modifier (adaptation à de nouvelles machines ou modification des équations du modèle).

Pour remédier à cela, le CAPS a décidé d'utiliser des opérateurs numériques discrets. Le code conserve une forme proche des équations et on évite les lignes interminables composées de manipulations d'indices de tableaux. Cela a été rendu possible par les progrès des compilateurs qui effectuent maintenant automatiquement l'expansion des sous-programmes et fonctions quand cela est utile.

La résolution des équations hydro-dynamiques peut se faire en quatre étapes :

• La première étape consiste à écrire l'équation continue, par exemple :

$$\frac{\partial T}{\partial t} = -\frac{\partial (uT)}{\partial x} - \frac{\partial (vT)}{\partial y} - \frac{\partial (wT)}{\partial z}$$

Ici, $T = T(x, y, z, t)$ est un scalaire, t représente le temps, u, v et w sont les composantes de la vitesse dans les directions x, y et z respectivement. On remarque que les

calculs des trois termes du second membre sont indépendants, cela peut être utilisé sur une machine parallèle.

• La deuxième étape consiste à écrire les équations sous forme discrète appropriée. Par exemple,

$$\overline{A(x)}^{nx} = \frac{A\left(x + \frac{n\Delta x}{2}\right) + A\left(x - \frac{n\Delta x}{2}\right)}{2}$$

$$\delta_{nx} A(x) = \frac{A\left(x + \frac{n\Delta x}{2}\right) - A\left(x - \frac{n\Delta x}{2}\right)}{n\Delta x}$$

Ces opérateurs discrets représentent respectivement un opérateur de moyenne (interpolation) et de dérivée du premier ordre. Ils peuvent être combinés pour former d'autres opérateurs plus complexes. On applique ces opérateurs à l'équation précédente sur une grille Arakawa-C avec un schéma saute-mouton d'ordre deux en temps :

$$\delta_{2t} T^n_{i,j,k} = -\delta_x(u\overline{T}^x)^n_{i,j,k} - \delta_x(v\overline{T}^y)^n_{i,j,k} - \delta_x(w\overline{T}^z)^n_{i,j,k}$$

où les indices i, j, k correspondent aux directions x, y, et z (par exemple, $x_i = i\Delta x$), l'indice supérieur n est l'indice correspondant au pas de temps $t = n\Delta t$. Jusque là, la structure des équations est conservée.

• La troisième étape consiste à développer cette expression, ce qui nous donne :

$$\begin{aligned}
\frac{T^{n+1}_{i,j,k} - T^{n-1}_{i,j,k}}{2\Delta t} = & -\frac{1}{\Delta x}\left[u^n_{i,j,k}\frac{T^n_{i+1,j,k} + T^n_{i,j,k}}{2} - u^n_{i-1,j,k}\frac{T^n_{i,j,k} + T^n_{i-1,j,k}}{2}\right] \\
& -\frac{1}{\Delta y}\left[v^n_{i,j,k}\frac{T^n_{i,j+1,k} + T^n_{i,j,k}}{2} - v^n_{i,j-1,k}\frac{T^n_{i,j,k} + T^n_{i,j-1,k}}{2}\right] \\
& -\frac{1}{\Delta z}\left[w^n_{i,j,k}\frac{T^n_{i,j,k+1} + T^n_{i,j,k}}{2} - w^n_{i,j,k-1}\frac{T^n_{i,j,k} + T^n_{i,j,k-1}}{2}\right]
\end{aligned}$$

• Enfin, cette expression est traduite dans un langage de programmation. Pour notre exemple, une programmation en Fortran serait :

```
DO k=1,nz
  DO j=1,ny
    DO i=1,nx
      T(i,j,k,n+1) = T(i,j,k,n-1)
        - 0.5*dt/dx*( u(i,j,k,n)*(T(i+1,j,k,n)+T(i,j,k,n)) -
                      u(i-1,j,k,n)*(T(i,j,k,n)+T(i-1,j,k,n)))
        - 0.5*dt/dy*( v(i,j,k,n)*(T(i,j+1,k,n)+T(i,j,k,n)) -
                      v(i,j-1,k,n)*(T(i,j,k,n)+T(i,j-1,k,n)))
        - 0.5*dt/dz*( w(i,j,k,n)*(T(i,j,k+1,n)+T(i,j,k,n)) -
                      w(i,j,k-1,n)*(T(i,j,k,n)+T(i,j,k-1,n)))
    ENDDO
```

```
  ENDDO
ENDDO
```

On constate que cette forme est très éloignée de la structure des équations de départ et qu'elle est assez peu lisible. Il est par exemple difficile de repérer les trois différenciations indépendantes. Cela rend plus difficile les modifications du code et peut entraîner des erreurs dues aux nombreuses manipulations d'indices.

La solution adoptée consiste à éliminer l'étape d'expansion des opérateurs et à utiliser la structure d'opérateurs dans le code. On définit des sous-programmes qui effectuent les opérations correspondant aux opérateurs cités plus haut. Chaque routine reçoit en entrée une variable et éventuellement des informations concernant sa dimension et sa place sur la grille et retourne la variable transformée résultant de l'opération effectuée.

Par exemple, soit `AVGX ( var- entrée, var- sortie )` et `DIFFX ( var- entrée, var-sortie)` les sous-programmes effectuant les opérations de moyenne et de différenciation dans la direction x (les mêmes opérations existent dans les directions y et z). On défini aussi `AAMULT(entrée1, entrée2, sortie)` qui rend le produit terme à terme des matrices `entrée1` et `entrée2`. Le code précédent s'écrit alors :

```
CALL avgx(T, temp1)                ! temp1 reçoit T^x
CALL avgy(T, temp2)                ! temp2 reçoit T^y
CALL avgz(T, temp3)                ! temp3 reçoit T^z

CALL aamult(u, temp1, temp1)       ! temp1 reçoit UT^x
CALL aamult(v, temp2, temp2)       ! temp2 reçoit VT^y
CALL aamult(w, temp3, temp3)       ! temp3 reçoit WT^z

CALL diffx(temp1, temp1)           ! temp1 reçoit δx(UT^x)
CALL diffx(temp2, temp2)           ! temp2 reçoit δy(VT^y)
CALL diffx(temp3, temp3)           ! temp3 reçoit δz(WT^z)

DO k=1,nz
  DO j=1,ny
    DO i=1,nx
      T(i,j,k,n+1)=T(i,j,k,n-1) - 2.*dt*
        ( temp1(i,j,k,n)+temp2(i,j,k,n)+temp3(i,j,k,n) )
    ENDDO
  ENDDO
ENDDO
```

On constate que la structure des trois types d'opérations effectuées est nettement visible, les opérations dans chaque direction peuvent être effectuées indépendamment. A partir du moment où les opérateurs ont été vérifiés et validés, il est facile

de vérifier l'exactitude du code et de le modifier. Toutes les manipulations d'indices ont lieu dans les routines de plus bas niveau (qui contiennent au maximum 10 lignes de code) et sont cachées à l'utilisateur. On verra que c'est en partie à ce niveau que se situe le travail de parallélisation.

6.3 Parallélisation

L'approche que nous avons retenue pour paralléliser ARPS est la décomposition de domaine. Le domaine de calcul est découpé en sous-domaines selon les trois directions de l'espace. Chaque sous-domaine est affecté à un processeur qui effectue l'intégration du modèle dans ce sous-domaine. La discrétisation de ARPS est telle qu'à un pas de temps donné, les seules informations extérieures au sous-domaine dont on a besoin sont les valeurs sur les frontières des sous-domaines adjacents. Ces valeurs sont utilisées pour calculer les valeurs des dérivées intervenant dans les équations d'évolution pour les points aux bords du sous-domaine (voir figure 6.4). Ces valeurs se trouvent à chaque pas de temps dans la mémoire locale du processeur adjacent. Elles peuvent être envoyées au processeur courant en utilisant une bibliothèque d'échange de messages.

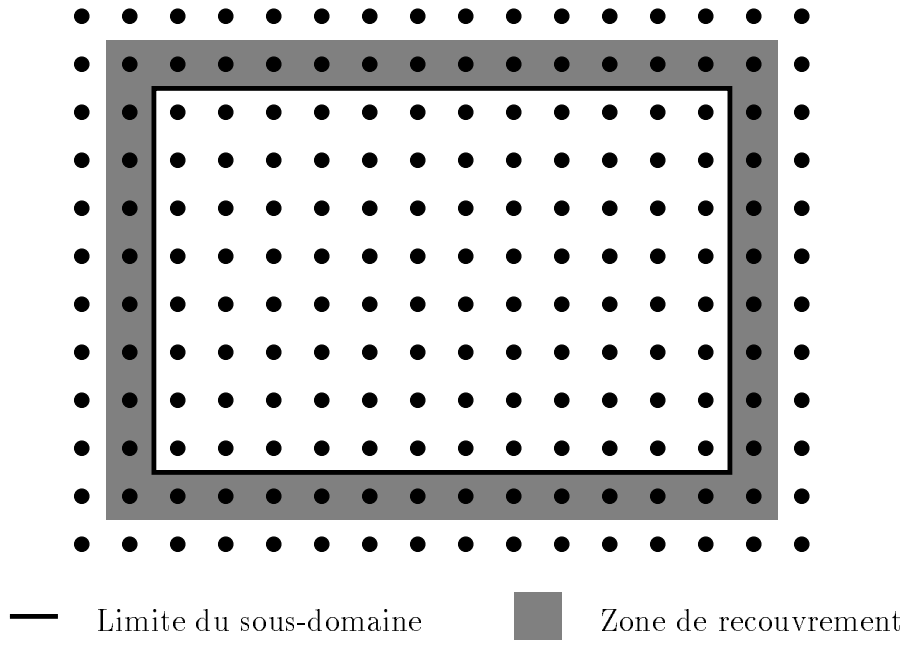


FIG. 6.4 - *Décomposition de domaine.*

Il existe plusieurs stratégies pour diviser la grille du modèle en sous-domaines. Une idée « naturelle » est de partitionner le domaine de manière à minimiser la surface des sous-domaines relativement à leur volume. Cela permet de réduire le temps de communication (proportionnel à la surface) à un niveau aussi bas que possible

par rapport au temps de calcul (proportionnel au volume). Un bon compromis est obtenu en utilisant des sous-domaines parallélépipèdiques (la forme optimale est la boule mais cela pose des problèmes de recouvrement et de raccord). Le nombre de découpage dans chaque direction est un paramètre du programme, on le modifiera pour utiliser au mieux la machine parallèle dont on dispose.

Sur chaque sous-domaine, le code séquentiel est utilisé avec très peu de modifications qui sont :

- La première modification consiste à changer les dimensions des tableaux du programme afin de stocker les valeurs des frontières internes aux sous-domaines adjacents.

- La deuxième modification à apporter est de modifier les indices de boucles de manière à prendre en compte ces valeurs. La difficulté essentielle de ceci est que cette modification est différente selon que la boucle concerne une valeur vectorielle ou une valeur scalaire à cause de la structure de la grille « Arakawa C ».

- La troisième modification consiste à ajouter les envois et réceptions de messages pour l'échange de valeurs entre sous-domaines entre chaque pas de temps. Deux types d'échanges sont nécessaires : un pour les petits pas de temps (partie acoustique du code) et un pour les grands pas de temps.

- Enfin, il faut assurer la distribution des données au début du calcul et le regroupement des résultats à la fin du calcul.

6.3.1 Essais et performances

Les résultats présentés ont été obtenus sur un IBM SP1 en utilisant la bibliothèque d'échange de messages PVM (voir 2.3).

Référence : le code séquentiel

L'exécution séquentielle qui sert de référence pour mesurer l'efficacité de la parallélisation a été faite sur un nœud de l'IBM SP1. La taille de la grille utilisée est de $64 \times 64 \times 32$. Une simulation de 300 secondes avec un petit pas de temps de 1s et un grand pas de temps de 6s s'exécute en 1869 secondes de calcul auxquelles il faut ajouter 126 secondes d'opérations d'entrées/sorties.

On peut remarquer dans le tableau 6.1 que dans le code séquentiel, il existe un problème de défaut de cache lorsque le nombre de points est 32 dans toutes les directions [1] (sur RS6000 modèle 530 pour ARPS avec 11 pas de temps). En fait, pour

1. Après une discussion avec K. Johnson et K. Droegemeier, il s'avère que ce problème apparaît plus généralement lorsque le code est utilisé avec un nombre de points qui est une puissance de 2.

être sûr de ne pas avoir de problème, il faut prendre au moins l'une des dimensions qui ne soit pas une puissance de 2.

Taille de la grille	Temps total d'exécution en secondes
28x28x28	63.66
30x30x30	68.56
31x31x31	78.78
31x31x32	80.66
32x32x31	84.36
32x31x32	84.29
31x32x32	84.48
32x32x32	220.47
33x33x33	92.18

TAB. 6.1 - *Temps d'exécution de ARPS sur IBM RS6000.*

Performances du code parallèle

Ce travail a été fait avec la bibliothèque d'échange de messages PVM [26]. Les résultats présentés dans le tableau 6.2 ont été obtenus sur l'IBM SP1 de l'IMAG. Le temps total représente le temps d'exécution vu par l'utilisateur, le temps CPU est le temps de calcul du processeur le plus chargé. Tous ces temps sont obtenus sur une grille de taille 64x64x23 et correspondent à un temps simulé de 150 secondes.

Processeurs	Temps total (sec.)	Temps CPU (sec.)
1	343	274
4	89	71
9	92	60

TAB. 6.2 - *Temps d'exécution sur IBM SP1 en utilisant le switch.*

Les performances sont excellentes pour un petit nombre de processeurs mais se dégradent vite quand le nombre de processeurs augmente. Deux facteurs sont en cause :

- Les mesures présentées ont été réalisées avec une taille de problème fixée. Quand le nombre de processeurs augmente, les tâches de calcul deviennent petites par rapport au volume de communication. Il faudrait réaliser des mesures ou la taille du problème augmente avec le nombre de processeurs pour garder une granularité suffisante.

- La parallélisation du code a été conçue dans l'optique de recouvrir les communications par des calculs. Malheureusement le SP1 n'offre pas cette fonctionnalité.

On peut aussi s'intéresser aux performances du code selon le réseau de communication et les options de PVM que l'on utilise. Le tableau 6.3 donne les temps d'exécution de ARPS dans la configuration déjà utilisée dans l'essai précédent sur 4 processeurs.

Mode de communication	Temps total (sec.)	Temps CPU (sec.)
Défaut + Ethernet	662	221
Défaut + Switch	381	231
Raw + RouteDirect + Switch	89	71

TAB. 6.3 - *Comparaison des deux réseaux du SP1.*

On constate une différence de performances très nette selon le réseau et les options de PVM que l'on utilise. L'écart de temps CPU selon que l'on utilise ou non l'option `PVMRAW` (voir chapitre 2.3) est le temps correspondant au codage XDR des données, il n'est pas du tout négligeable. On constate également la supériorité du switch sur le réseau ethernet malgré l'utilisation du protocole IP.

6.3.2 Conclusion

A cause de la grande modularité du code et de l'utilisation d'opérateurs discrets pour effectuer les opérations mathématiques de base (dérivations, moyennes, etc...), la structure mathématique des équations continues est, au moins en partie, préservée, ce qui procure une plus grande facilité d'apprentissage, de modification et de débogage. La présence de la documentation interne et externe a été d'une grande utilité pour apprendre le code et comprendre sa structure globale. Ces deux facteurs sont très importants pour l'écriture d'une version parallèle efficace. On peut seulement regretter que les limites d'application des opérateurs discrets (c'est-à-dire des boucles qu'ils comportent) ne soit pas toujours un paramètre des procédures correspondantes.

On peut retenir de tout ceci qu'avec un investissement « raisonnable », on peut tirer de bonnes performances d'une machine parallèle. Comme cela a déjà été écrit plus haut, ceci n'est possible que pour un code qui respecte certaines règles d'écriture qui procurent une bonne lisibilité comme c'est le cas pour ARPS.

On notera toutefois que toutes les mesures présentées ici ne prennent pas en compte les entrées-sorties du programme. Si on en tient compte, les performances du programme parallèle s'écroulent de manière spectaculaire, les entrées-sorties parallèles pouvant même être plus lentes que la même opération sur une machine séquentielle. C'est dans cette direction que le plus gros travail reste à faire pour les applications de type météorologique où ce qui compte est la rapidité de la chaîne complète de traitement (réception des données, intégration du modèle et diffusion des résultats).

6.4 Perspectives

Le travail présenté ici constitue une première approche de ce qu'il est possible de faire avec un modèle tel que ARPS. La version 4.0 de ARPS disponible depuis Septembre 1995 comprend un certain nombre d'améliorations par rapport à la version 3.2 utilisée dans ce chapitre dont une version parallèle pour PVM ou MPI. L'adjoint de ce modèle est également disponible au CAPS. Tous les outils sont donc disponibles pour tester avec ce modèle des algorithmes parallèles d'assimilation de données.

Conclusion

Dans les sciences environnementales, le problème de l'assimilation de données devient de plus en plus crucial. Le coût à payer pour le résoudre est d'un ordre de grandeur supérieur au coût de la prévision proprement dite. La parallélisation de ce type d'algorithme est donc la clé de développements futurs (pollution, évolution du climat, ...) à cause des limitations aussi bien en terme de mémoire que de vitesse de calcul des super-calculateurs vectoriels.

Nous avons proposé dans ce document deux approches pouvant donner de bons résultats pour paralléliser le processus d'assimilation de données variationnelle. La première consiste à paralléliser le modèle utilisé ainsi que son adjoint du premier et éventuellement du second ordre. On utilise alors un algorithme d'optimisation sans contraintes classique pour mener à bien l'assimilation. Cette approche n'est pas bien adaptée à un modèle simple comme le modèle de Shallow water mais s'adapte beaucoup mieux aux modèles réels.

La deuxième approche consiste à décomposer le modèle sur plusieurs sous-domaines au sens large et à utiliser le modèle adjoint pour contrôler les conditions aux interfaces entre ces sous-domaines. Nous avons vu que selon les cas, cette approche peut s'avérer très efficace ou au contraire assez décevante. Cela tient en grande partie à la « forme » des sous-domaines et des interfaces qui les relient. En résumé, on peut dire que pour que cette méthode fonctionne bien, il faut que la dimension de la variable de contrôle aux interfaces reste raisonnable par rapport à la dimension de la variable de contrôle originale sur le même sous-domaine.

Les essais que nous avons effectués nous ont également appris, notamment grâce aux différentes représentations des exécutions parallèles, que d'importants problèmes d'équilibre de charge se posent. Le modèle utilisé ici est très simple et ne comporte pas en lui-même de source de déséquilibre, on peut donc supposer que ces problèmes vont s'amplifier lorsqu'on voudra utiliser cet algorithme avec un modèle plus réaliste qui comportera une modélisation de la micro-physique elle aussi mal équilibrée. Afin d'améliorer les résultats donnés ici, il serait nécessaire d'étudier une version asynchrone de l'algorithme incluant un mécanisme de régulation dynamique de la charge.

En ce qui concerne l'aspect pratique, on peut retenir que pendant les trois ans que ce travail a duré, les choses ont évolué de manière significative. Des outils de

mise au point (débogage et visualisation) de programmes parallèles apparaissent et sont d'une grande utilité. On pourrait aujourd'hui tenter une approche différente. Une étude intéressante pourrait être menée en comparant les résultats obtenus à ceux obtenus avec les outils de parallélisation qui apparaissent.

Néanmoins, en utilisant un paralléliseur automatique, on se prive d'algorithmes spécifiques nouveaux encore à découvrir. En effet, un paralléliseur « automatique » pourra au mieux ré-organiser les calculs d'un algorithme programmé classiquement. Il ne sera pas capable de changer d'algorithme s'il en existe qui s'adaptent mieux au calcul parallèle. De plus, le travail de conception d'algorithmes parallèles apporte un nouvel éclairage sur les méthodes connues et permet de faire émerger des algorithmes nouveaux : on peut faire mieux que paralléliser petit à petit un programme séquentiel. Cela implique bien sûr des connaissances dans le domaine de la programmation parallèle mais aussi dans le champ spécifique de l'application. Cela restera vrai jusqu'au développement de compilateur-paralléliseurs performants et même ensuite car ceux-ci ne pourront pas créer de toute pièce les algorithmes parallèles qui restent à découvrir...

La première extension de ce travail serait l'assimilation de données en parallèle sur le modèle ARPS. Nous avons vu que la dernière version de ce modèle (ARPS 4.0, voir [15]) comprend une extension parallèle utilisant PVM ou MPI et que son adjoint est disponible. Malheureusement, à l'heure ou nous écrivons ces lignes, elle n'est disponible que depuis 10 jours, nous n'avons donc pas encore pu effectuer ce travail.

La deuxième perspective, à plus long terme, est l'assimilation de données complète et consistante sur les deux fluides géophysiques. Les problèmes qui se posent sont dus aux différentes propriétés physiques des deux fluides. Les temps caractéristiques qui les régissent sont différents : l'évolution de l'atmosphère est beaucoup plus rapide que celle de l'océan (quelques heures à quelques jours contre des semaines voire des mois). D'autre part, les quantités d'information disponible ne sont pas les mêmes. En météorologie, on dispose de nombreuses données provenant de satellites et de stations d'observations. Ces données sont réparties dans l'espace. En océanographie, les données sont moins nombreuses et sont localisées sur la surface de l'océan.

Bibliographie

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*, 1992.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK : A Portable Linear Algebra Library for High-Performance Computers. Working Note, Argonne National Laboratory, May 1990. LAPACK Working Note 20.

[3] G. Authié, A. Ferreira, J.-L. Roch, G. Villard, J. Roman, C. Roucairol, and B. Virot, editors. *Algorithmes parallèles, analyse et conception.* Hermès, 1994.

[4] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, New York, 1994.

[5] V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C.-T. Ho, G. Irwin, S. Kipnis, R. Laurence, and M. Snir. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4):445–462, April 1994. Special issue: Message Passing Interfaces.

[6] J. Bauer. PVM Instrumentation and ARPS 2.0. Technical report, CAPS, December 1991.

[7] A. Bensoussan, J. L. Lions, and R. Temam. Sur les méthodes de décomposition, de décentralisation et de coordination et applications. In *Cahier IRIA*, volume 11, 1972.

[8] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation, Numerical Methods.* Prentice-Hall, 1989.

[9] R. M. Butler and E. L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994. Special issue: Message Passing Interfaces.

[10] Y. Cai and I. M. Navon. Parallel block preconditioning technics for the numerical simulation of shallow water flow using finite-element methods. *Journal of Computational Physics*, 122:39–50, November 1995.

[11] R. Calkin, R. Hempel, H.-C. Hoppe, and P. Wypior. Portable programming with the Parmacs message-passing library. *Parallel Computing*, 20(4):615–632, April 1994. Special issue: Message Passing Interfaces.

[12] C. Calvin and L. Colombet. Introduction au modèle de programmation par processus communicants : deux exemples PVM et MPI. Rapport Apache 12, Institut IMAG, July 1994.

[13] C. Calvin and L. Colombet. Performance Evaluation and Modeling of Collective Communications on Cray T3D. *Submitted to Parallel Computing*, 1995.

[14] Center for Analysis and Prediction of Storms. *ARPS 3.0 User's Guide*, October 1992. University of Oklahoma,National Science Foundation.

[15] Center for Analysis and Prediction of Storms. *ARPS 4.0 User's Guide*, September 1995. University of Oklahoma.

[16] P. G. Ciarlet. *Introduction à l'analyse numérique matricielle et à l'optimisation.* Masson, 1988.

[17] L. Colombet, L. Desbat, L. Gautier, F. Ménard, Y. Trémolet, and D. Trystram. Industrial and Scientific Applications using PVM. In *Parallel CDF'93*, 1993.

[18] Cray. *The Cray MPP Fortran Reference Manual*, 1994.

[19] R. Daley. Atmospheric data assimilation. *The Geophysical Magazine*, 1, 1995. Series 2.

[20] R. S. Demboand and T. Steihaug. Truncated-Newton Algorithms for Large-Scale Unconstrained Optimization. *Mathematical Programming*, pages 190–212, 1983.

[21] J. Demmel, J. Dongarra, and W. Kahan. On Designing Portable High Performance Numerical Libraries. Working Note CS-91-141, University of Tennessee, Computer Science Department, July 1991. Lapack Working Note 39.

[22] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice Hall, 1983.

[23] C. Farhat and F.-X. Roux. Implicit parallel processing in computational mechanics. *Computational Mechanics Advances*, 2(1), 1994.

[24] J. Flower and A. Kolawa. Express is not just a message passing system: Current and future directions in Express. *Parallel Computing*, 20(4):597–614, April 1994. Special issue: Message Passing Interfaces.

[25] I. Foster. *Designing and Bulding Parallel Programs.* Addison-Wesley, 1995.

[26] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, May 1994.

[27] 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.* The MIT Press, 1994.

[28] G. Geist, J. Kohl, R. Manchek, and P. Papadopoulos. New features of pvm 3.4 and beyond. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *EuroPVM'95.* Hermes, September 1995.

[29] M. Ghil and P. Malanotte Rizzoli. Data assimilation in meteorology and oceanography. *Advances in Geophysics*, 33:141–266, 1991.

[30] J.-C. Gilbert and C. Lemaréchal. Some numerical experiments with variable storage quasi-Newton algorithms. *Mathematical Programming*, 45:407–435, 1989.

[31] F. Guinand. *Ordonnancement avec communications pour architectures multiprocesseurs dans divers modèles d'exécution.* PhD thesis, Institut National Polytechnique de Grenoble, 1995.

[32] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[33] HPF Forum. *High Performance Fortran Language Specification*, May 1993.

[34] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[35] IBM. *IBM AIX PVMe User's guide and Subroutine Reference*, April 1994.

[36] R. E. Kalman. A new approach to linear filtering and prediction problems. *J. Basic Eng.*, 82D:35–45, 1960.

[37] R. E. Kalman and R. S. Bucy. New results in linear filtering and prediction theory. *J. Basic Eng.*, 83D:95–108, 1961.

[38] T. Kauranne and G.-R. Hoffmann. Parallel Processing: A View From ECMWF. In *Fourth Workshop on Use of Parallel Processors in Meteorology*, pages 148–169. European Center for Medium Range Weather Forecasts, November 1992.

[39] F.-X. LeDimet. Determination of the adjoint of a numerical weather prediction model. Technical report, Florida State University, 1998.

[40] F. X. LeDimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations : Theorical aspects. *Tellus*, 38A:97–110, 1986.

[41] P. Lemonnier. Résolution numérique d'équations aux dérivées partielles par décomposition et coordination. In *Cahier IRIA*, volume 11, 1972.

[42] P. LeTallec. Domain decomposition methods in computational mechanics. *Computational Mechanics Advances*, 1(2):121–220, 1994.

[43] J. L. Lions. *Contrôle optimal de systèmes gouvernés par des équations aux dérivées partielles*. Dunod, 1968.

[44] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

[45] E. Lorenz. Deterministic nonperiodic flow. *J. Atmos. Sci.*, 20:130–141, 1963.

[46] E. Maillet. TAPE/PVM an efficient performance monitor for PVM applications – User Guide. Technical report, IMAG, Grenoble, 1995. Available by anonymous ftp from *ftp.imag.fr* in */pub/APACHE/TAPE*.

[47] O. A. McBryan. An overview of message passing environments. *Parallel Computing*, 20(4):417–444, April 1994. Special issue: Message Passing Interfaces.

[48] C. Mead and L. Conway. *Introduction aux systèmes VLSI*. InterEditions, 1983.

[49] MPI Forum. *MPI: A Message Passing Interface Standard*, April 1994.

[50] I. M. Navon, P. K. H. Phua, and M. Ramamurthy. Vectorization of conjugate gradient for large scale minimization. *Journal of Optimization Theory and its Applications*, 66:71–94, 1990.

[51] I.M. Navon and Y. Cai. Domain Decomposition and Parallel Processing of a Finite Element Model of the Shallow Water Equations. *Computer Methods in Applied Mechanics and Engineering*, 106:179–212, June 1993.

[52] Oak Ridge National Laboratory. *Paragraph User's Guide*.

[53] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4):463–480, April 1994. Special issue: Message Passing Interfaces.

[54] B. Plateau. Apache: Algorithmique parallèle et partage de charge. Rapport Apache 1, Institut IMAG, November 1994.

[55] D. A. Reed. Performance instrumentation techniques for parallel systems. *LNCS*, 729:463–490, 1993.

[56] N. Rostaing-Schmidt. *Différentiation automatique: application à un problème d'optimisation en météorologie*. PhD thesis, Université de Nice-Sophia Antipolis, December 1993.

[57] Rumeur. *Communications dans les réseaux de processeurs*. Collection ERI. Masson, 1994.

[58] C. Stunkel, D. Shea, D. Grice, P. Hochschild, and M. Tsao. The SP1 high performance switch. In *Scalable High-Performance Computing Conference*, pages 150–157, 1994.

[59] V. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, April 1994. Special issue: Message Passing Interfaces.

[60] A. Tanenbaum. *Architecture de l'ordinateur*. InterEditions, 1991.

[61] Y. Trémolet and F. X. Le Dimet. Parallel Data Assimilation in Meteorology. *European Geophysical Society XIXe General Assembly*, 1994.

[62] Y. Trémolet, F. X. Le Dimet, and D. Trystram. Parallelization of Scientific Applications : Data Assimilation in Meteorology. In *High Performance Computing and Networking*, volume 796 of *Lecture Notes in Computer Science*, 1994.

[63] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994. Special issue: Message Passing Interfaces.

[64] Z. Wang, I. M. Navon, F. X. Le Dimet, and X. Zou. The Second Order Adjoint Analysis : Theory and Applications. *Meteorology and atmospheric physics*, 1992.

[65] X. Zou, I. M. Navon, M. Berger, K. H. Phua, T. Schlick, and F. X. Le Dimet. Numerical Experience with Limited-Memory Quasi-Newton Methods. *SIAM J. Optimization*, 3(3):582–608, August 1993.

Le problème de l'assimilation de données sous sa forme générale peut se formuler : « comment utiliser simultanément un modèle théorique et des observations pour obtenir la meilleure prévision météorologique ou océanographique? ». Sa résolution est très coûteuse : pour la prochaine génération de modèles elle nécessitera une puissance de calcul de l'ordre de 10 Tflops. À l'heure actuelle, aucun calculateur n'est capable de fournir de telles performances mais cela devrait être possible dans quelques années, en particulier grâce aux ordinateurs parallèles à mémoire distribuée. Mais la programmation de ces machines reste un processus compliqué et on ne connaît pas de méthode générale pour paralléliser de manière optimale un algorithme donné. Nous tenterons de répondre au problème de la parallélisation de l'assimilation de données variationnelle, ce qui nous conduira à étudier la parallélisation d'algorithmes numériques d'optimisation assez généraux.

Pour cela, nous étendrons la méthodologie de l'écriture des modèles adjoints au cas où le modèle direct est parallèle avec échanges de messages explicites. Nous étudierons les différentes approches possibles pour paralléliser la résolution du problème de l'assimilation de données: au niveau des modèles météorologiques direct et adjoints, au niveau de l'algorithme d'optimisation ou enfin au niveau du problème lui-même. Cela nous conduira à transformer un problème séquentiel d'optimisation sans contraintes en un ensemble de problèmes d'optimisation relativement indépendants qui pourront être résolus en parallèle. Nous étudierons plusieurs variantes de ces trois approches très générales et leur utilité dans le cadre du problème de l'assimilation de données. Nous terminerons par l'application des méthodes de parallélisation précédentes au modèle de Shallow Water et comparerons leurs performances. Nous présenterons également une parallélisation du modèle météorologique ARPS (Advanced Regional Prediction System).

Mots clés : assimilation de données, algorithmique parallèle, modèles adjoints, contrôle optimal, optimisation.

The data assimilation problem has the general form : "how to simultaneously use a theoretical model and some observations to obtain the best meteorological or oceanographical prediction ?". This process is very expensive, for the next model generation, one may estimate that it will require a computational power of 10 Tflops. Nowadays, no computer can achieve such performances but it should be possible in a few years using parallel distributed memory computers. Even programming these machines is still not a straightforward process and no general method is known to obtain an optimal parallelization of any given algorithm, we shall try to answer the problem of parallelizing the variational data assimilation algorithm. This will lead us to consider parallelization of numerical unconstrained optimization algorithms which are quite general.

In this respect, we extend the adjoint writing methodology to the case of a parallel model using explicit message passing. We study the different possible approaches in order to parallelize the solution of the data assimilation problem : at the direct and adjoint models level, at the optimization algorithm level or at the data assimilation level itself. This implies to transform a sequential unconstrained optimization problem into a set of optimization problem relatively independent which can be solved in parallel. We study several variations on these three very general approaches and their usefulness for data assimilation. We finish by applying these methods to the Shallow Water model and by comparing their performances. We also present the ARPS model (Advanced Regional Prediction System) and its parallelization.

Keywords : data assimilation, parallel algorithms, adjoint models, optimal control, optimization.