



Asynchronisme : Cadres continu et discret

Sylvain Contassot-Vivier

► To cite this version:

Sylvain Contassot-Vivier. Asynchronisme : Cadres continu et discret. Calcul parallèle, distribué et partagé [cs.DC]. Université de Franche-Comté, 2006. tel-01463155

HAL Id: tel-01463155

<https://inria.hal.science/tel-01463155>

Submitted on 9 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives| 4.0 International License

**HABILITATION À DIRIGER DES RECHERCHES
UNIVERSITÉ DE FRANCHE-COMTÉ**

Spécialité

Informatique

présentée par

Sylvain CONTASSOT-VIVIER

Docteur en Informatique

Sujet

Asynchronisme : Cadres continu et discret

Soutenue le **1er Décembre 2006** devant le jury

Rapporteurs

Frédéric Desprez	Directeur de Recherche INRIA, ENS Lyon
Mohamed Ould-Khaoua	Reader, Université de Glasgow, Écosse
Jean Roman	Professeur, Université Bordeaux 1

Examineurs

Jacques Bahi	Professeur, Université de Franche-Comté
Christian Michel	Professeur, Université Louis Pasteur Strasbourg
Serge Miguet	Professeur, Université Lumière Lyon 2

À mon épouse, Alexandra,

À mes parents,

Remerciements

Je remercie les membres du jury, qui m'ont fait l'honneur de bien vouloir évaluer mes travaux de recherche.

Je tiens à remercier chaleureusement tous les membres de l'équipe *Algorithmique Numérique Distribuée* avec qui j'ai eu la chance de partager plusieurs années de *vie commune* dans une ambiance amicale et détendue.

J'adresse un remerciement supplémentaire à ceux qui ont partagé mon bureau durant toutes ces années.

Un grand merci aussi aux personnes avec qui j'ai directement travaillé, pour leur professionnalisme et leur ouverture d'esprit.

Enfin, je remercie particulièrement le Professeur Jacques Bahi pour la confiance qu'il m'a accordée dès le début de notre collaboration et pour m'avoir fait bénéficier sans retenue de son expérience dans le métier de la recherche.

Qu'il soit assuré de ma reconnaissance sincère.

Table des matières

Introduction	1
1 Formalisme des systèmes dynamiques	5
1.1 Dynamiques synchrones	6
1.2 Dynamiques asynchrones	8
2 Systèmes dynamiques continus asynchrones	11
2.1 Formalisme spécifique	11
2.1.1 Convergence pour les problèmes linéaires	12
2.1.2 Convergence pour les problèmes non linéaires	12
2.2 État de l’art	13
2.2.1 Classification des algorithmes itératifs parallèles	14
2.3 Intérêt général de l’asynchronisme	19
2.3.1 Schéma algorithmique général	20
2.3.2 Problème linéaire creux stationnaire	26
2.3.3 Problème non linéaire non stationnaire	29
2.3.4 Discussion sur la notion d’accélération dans les grappes hétérogènes .	33
2.4 Équilibrage de charge	35
2.4.1 Équilibrage statique	36
2.4.2 Équilibrage dynamique	37
2.5 Détection décentralisée de la convergence globale	50
2.5.1 Version théorique	51

2.5.2	Version pratique	54
2.6	Environnements de développement	59
2.6.1	Environnements testés	60
2.6.2	Performances	61
2.6.3	Programmation	64
2.6.4	Déploiement	65
2.6.5	Caractéristiques de développement des AIAC	66
2.7	Conclusion et perspectives	66
3	Systèmes dynamiques discrets asynchrones	69
3.1	Formalisme spécifique	70
3.1.1	Propriétés de l'évolution asynchrone	70
3.1.2	Définitions et outils	73
3.2	État de l'art	75
3.3	Convergence basée sur des propriétés locales	76
3.3.1	Notions supplémentaires	77
3.3.2	Résultat théorique	77
3.3.3	Algorithme de test	79
3.3.4	Application à un réseau booléen de Hopfield	82
3.4	Mixage synchronisme/asynchronisme	85
3.4.1	Stratégie de mixage	86
3.4.2	Application	87
3.5	Conclusion et perspectives	90
	Conclusion	91
	Liste des publications	93
	Bibliographie	99

Table des figures

2.1	Schéma d'exécution d'un algorithme SISC avec deux processeurs. Les blocs grisés représentent les itérations et les flèches représentent les communications de données. Les parties horizontales des flèches indiquent les mises en attentes des communications dues aux synchronisations.	15
2.2	Schéma d'exécution d'un algorithme SIAC avec deux processeurs et des envois de données en deux groupes. Les flèches discontinues représentent les envois du premier groupe de données et les flèches continues, celles du second groupe. Les parties horizontales des flèches indiquent les attentes d'incorporation dans les calculs des données reçues.	16
2.3	Schéma d'exécution d'un algorithme AIAC avec deux processeurs. Les parties horizontales des flèches indiquent les attentes d'incorporation dans les calculs des données reçues.	17
2.4	Schéma d'exécution d'un algorithme AIAC avec communications semi-flexibles sur les réceptions	18
2.5	Schéma d'exécution d'un algorithme AIAC avec communications semi-flexibles sur les envois en deux groupes de données. Les flèches discontinues représentent les envois du premier groupe de données et les flèches continues, celles du second groupe. Les parties horizontales des flèches indiquent les attentes d'incorporation dans les calculs des données reçues.	18
2.6	Schéma d'exécution d'un algorithme AIAC avec communications flexibles en deux groupes de données. Les flèches discontinues représentent les envois du premier groupe de données et les flèches continues, celles du second groupe.	19
2.7	Schéma d'exécution d'un algorithme AIAC avec communications semi-flexibles sur les réceptions et communications exclusives. Les flèches en pointillés représentent les communications qui ne sont pas effectuées à cause de l'exclusivité des communications.	22
2.8	Décomposition du problème linéaire pour quatre processeurs.	27

2.9	Processus itératif de résolution du problème linéaire sur le processeur 2. . . .	27
2.10	Organisation des données dans les tableaux AncY et NouvY sur le processeur i . . .	40
2.11	Schéma d'exécution d'un algorithme AIAC avec communications flexibles exclusives. Les flèches en pointillés représentent les communications non effectuées à cause de l'exclusivité des communications. Les flèches solides commençant en cours d'itération correspondent aux envois des données vers le voisin de gauche alors que les flèches commençant en fin d'itération correspondent aux envois des données vers le voisin de droite.	40
2.12	Schéma d'exécution de l'algorithme d'équilibrage entre deux processeurs. . . .	41
2.13	Transfert de données d'un processeur vers son voisin de gauche.	42
2.14	Transfert de charge entre deux processeurs, basé sur le résidu.	47
2.15	Temps d'exécution de l'algorithme AIAC de résolution du problème de Brusselator 1D équilibré avec deux estimateurs de charge différents en fonction de la précision demandée.	48
2.16	Temps d'exécution de l'algorithme AIAC de résolution du problème de Brusselator 1D avec ou sans équilibrage sur une grappe homogène locale en fonction du nombre de processeurs.	49
2.17	Processus de détection de la convergence globale basé sur l'élection de leader. Les chiffres indiquent le nombre de voisins dans l'arbre couvrant dont le processeur n'a pas encore reçu de message de convergence partielle.	52
2.18	Temps d'exécution de l'algorithme AIAC de résolution du problème d'advection-diffusion 2D sur une grappe locale hétérogène en fonction du nombre de processeurs pour chaque environnement.	64
3.1	Évolution synchrone avec retards à partir de l'état 011 menant à un état (100 ou 110) non atteignable dans une évolution synchrone sans retards. Les chiffres entre parenthèses indiquent les retards utilisés (* pour n'importe lequel). . . .	72
3.2	Évolution asynchrone à partir de l'état 011 menant à un état (100 ou 110) non atteignable dans une évolution chaotique. Les retards ne sont mentionnés que lors des mises à jour des éléments.	73
3.3	États supplémentaires (gris foncé) menant au point fixe $(0, 0, 0)$ autour de son voisinage contractant.	80
3.4	Décomposition par blocs conservant un cycle dans le graphe de connexion du nouveau système.	87
3.5	Graphe de connexion d'un système à six éléments.	87

3.6	Graphe d'itération de la fonction f en mode synchrone.	88
3.7	Graphe d'itération partiel de la fonction f en mode asynchrone.	89
3.8	Graphe de connexion du système décomposé par blocs.	89

Liste des tableaux

2.1	Temps de calcul (en secondes) de l'algorithme AIAC de résolution du problème linéaire creux sur une méta-grappe de dix machines réparties sur trois sites. .	28
2.2	Temps de calcul (en secondes) de l'algorithme AIAC de résolution du problème d'Akzo-Nobel avec deux organisations logiques d'une même méta-grappe de douze machines réparties sur quatre sites.	32
2.3	Performances de l'algorithme AIAC de résolution du problème d'Akzo-Nobel en séquentiel selon la puissance de la machine et en parallèle selon l'ordre des machines.	34
2.4	Temps d'exécution (en secondes) de l'algorithme AIAC de résolution du problème d'Akzo-Nobel avec ou sans équilibrage statique sur une méta-grappe hétérogène de huit machines réparties sur trois sites.	37
2.5	Temps d'exécution (en secondes) de l'algorithme AIAC de résolution du problème de Brusselator 1D avec ou sans équilibrage sur une méta-grappe hétérogène.	49
2.6	Temps d'exécution (en secondes) de l'algorithme AIAC de résolution du problème d'advection-diffusion 2D avec trois variantes de détection de la convergence globale.	59
2.7	Gestion des processus selon le problème traité et l'environnement utilisé (N est le nombre de processeurs).	61
2.8	Temps d'exécution (en secondes) de l'algorithme de résolution du problème linéaire creux sous différents environnements.	62
2.9	Temps d'exécution (en secondes) de l'algorithme de résolution du problème d'advection-diffusion sous différents environnements.	63
3.1	Table et représentation graphique de la fonction d'évolution f du système à trois éléments à valeurs dans $\{0, 1\}$	71
3.2	Fonction F du système discret à valeurs dans $E = \{0, 1, 2\}^3$	78

3.3	Ensemble des motifs mémorisés ($\circ \equiv 1$ et $\bullet \equiv 0$).	84
3.4	Résultats de la fonction de test et de la simulation pour huit initialisations du réseau ($\circ \equiv 1$ et $\bullet \equiv 0$).	84

Introduction

L'informatique est une science relativement jeune qui a connu plusieurs vagues d'évolution qui, en général, ont été étroitement liées aux progrès technologiques des composants des machines. On peut aisément considérer qu'une nouvelle étape d'évolution se situe au niveau des réseaux de communication qui commencent à atteindre une bonne qualité, tant au niveau de la fiabilité qu'au niveau des performances.

Au-delà de l'intérêt simplement pratique du transfert des données, cela implique une nouvelle vision de l'outil informatique pour le calcul scientifique intensif. En effet, après la vague des stations de travail, puis des machines parallèles et enfin des grappes de calcul sur réseaux locaux, cette amélioration des réseaux de communication permet d'envisager l'utilisation de *grappes de grappes*. Ce nouveau concept de *méta-grappe* est défini par un ensemble d'unités de calcul (stations de travail, grappe ou machine parallèle) réparties sur des sites géographiques différents. Ces méta-grappes sont donc généralement constituées de machines hétérogènes reliées par un réseau de communication connexe mais non complet dont les liens sont hétérogènes.

L'intérêt évident de telles méta-grappes est de permettre de rassembler un plus grand nombre de machines pour obtenir des traitements plus rapides et/ou traiter des problèmes de plus grande taille. En effet, l'ajout d'une machine à un système de calcul parallèle, même moins performante que celles qui y sont déjà, permet d'augmenter le nombre de traitements effectués en parallèle et donc d'en améliorer les performances. De même, un tel ajout permet d'augmenter la capacité mémoire globale du système et offre ainsi la possibilité de stocker et donc traiter des problèmes de plus grande taille. L'hétérogénéité des machines ne représente donc pas une limitation particulière des méta-grappes. De plus, sa gestion a déjà fait l'objet de plusieurs études dans le cadre des grappes locales de stations de travail. Par contre, un problème nouveau qui apparaît avec les méta-grappes de machines est la prise en compte efficace des liens de communications hétérogènes. Ce point n'a encore été que peu étudié.

Cependant, il faut noter que chaque évolution matérielle est souvent accompagnée d'une évolution logicielle. En effet, il est généralement nécessaire de modifier ou d'enrichir les paradigmes de programmation pour prendre en compte les nouvelles possibilités des machines. Le but étant clairement d'utiliser les nouvelles ressources de manière optimale ou quasi-optimale pour obtenir, soit un gain de qualité, soit un gain de temps, et si possible les deux. Ainsi, tout comme les machines parallèles et les grappes de stations ont induit

le développement des bibliothèques de communication dans les langages de programmation, l'apparition des grappes de grappes induit une mise à jour des paradigmes de programmation pour prendre en compte les spécificités de ces *méta-machines*.

En ce qui concerne la programmation parallèle, le modèle couramment utilisé est le passage de messages synchrones. Si ce modèle est tout à fait satisfaisant avec les machines parallèles de type SPMD (Single Program Multiple Data) ou les grappes de stations, il ne l'est plus avec les grappes de grappes. En effet, même si les communications distantes entre machines sont de plus en plus rapides, elles n'en restent pas moins bien plus lentes que les communications locales. Ainsi, si l'on programme une *méta-grappe* en utilisant des communications synchrones entre les processeurs, on s'aperçoit assez vite que les communications entre des machines distantes ralentissent considérablement le processus global de calcul.

Ainsi, il apparaît essentiel de modifier ce modèle de façon à pouvoir utiliser efficacement les méta-grappes. Or, il existe un autre mode de communication entre machines qui permet de se soustraire au moins partiellement aux contraintes des communications, c'est l'asynchronisme. Ce mode de communication repose principalement sur le fait que les communications entre processeurs ne bloquent pas le déroulement de l'algorithme. Ainsi, lors d'une communication, la machine émettrice n'attend pas que les données soient reçues sur la machine destinataire mais continue ses calculs locaux. De même, il n'y a pas d'attente explicite des réceptions des messages mais les données sont reçues au moment où elles arrivent. Cela permet donc de réaliser un recouvrement efficace des communications par des calculs et donc de réduire considérablement le coût des communications.

Malheureusement, ce type de communication n'est pas utilisable dans tous les types d'algorithmes. Cependant, il prend tout son sens avec les algorithmes itératifs. Contrairement aux méthodes directes qui donnent la solution d'un problème après un nombre fini d'opérations, les algorithmes itératifs procèdent par améliorations successives d'une approximation de la solution en répétant un processus donné. Lorsque l'approximation se rapproche effectivement de la solution, on dit qu'ils convergent vers celle-ci. Dans le cadre parallèle, ces algorithmes ont l'avantage de présenter une plus grande souplesse d'utilisation des communications. En effet, sous certaines conditions, pas trop restrictives, les données mises à jour ne sont plus systématiquement nécessaires à chaque sollicitation mais interviennent plus comme un facteur d'avancement du processus itératif. De plus, de nombreux problèmes scientifiques sont résolus par ce type d'algorithmes, notamment tous les problèmes d'EDP (Équations aux Dérivées Partielles), EDO (Équations Différentielles Ordinaires) et PVI (Problèmes à Valeurs Initiales). D'ailleurs, pour certains problèmes non-linéaires comme par exemple celui des racines de polynômes, les algorithmes itératifs sont les seules méthodes de résolution disponibles. Enfin, elles requièrent parfois moins de mémoire que les algorithmes directs, notamment pour les problèmes linéaires. On peut donc aisément évaluer l'intérêt de la mise en œuvre de tels algorithmes avec des communications asynchrones sur les méta-grappes.

Ce document présente l'ensemble des recherches que j'ai effectuées dans le domaine des algorithmes itératifs parallèles asynchrones. Le plan de ce document est divisé en trois grandes parties :

Formalisme des systèmes dynamiques : ce chapitre présente une description formelle générale des systèmes dynamiques ainsi que la méthode de décomposition utilisée pour leur parallélisation. De même, les différents modes d'évolution pouvant être utilisés sont décrits en se focalisant plus particulièrement sur les modes synchrone et totalement asynchrone.

Systèmes dynamiques asynchrones dans le cadre continu : ce chapitre présente le schéma général d'un algorithme itératif asynchrone pour la résolution de problèmes continus. Puis, une classification des différents types d'algorithmes parallèles itératifs est proposée pour permettre une identification précise des algorithmes étudiés dans mes recherches. Un rapide état de l'art sur les recherches effectuées dans le domaine de l'algorithmique parallèle itérative est donné. Enfin, mes travaux de recherche sur les algorithmes itératifs asynchrones sont présentés, aussi bien au niveau des méthodologies de mise en œuvre, notamment en ce qui concerne l'environnement de développement à utiliser, qu'au niveau algorithmique pour résoudre les différents problèmes spécifiques à ces algorithmes tels que la détection de la convergence, la procédure d'arrêt ou encore l'équilibrage de charge. Pour chacun de ces aspects, l'intérêt des choix effectués est évalué expérimentalement dans différents contextes d'utilisation généraux sur des grappes et méta-grappes de calcul.

Systèmes dynamiques asynchrones dans le cadre discret : ce chapitre présente dans un premier temps les spécificités du cadre discret et fournit une description détaillée des mécanismes intervenant dans le comportement de ces systèmes. Ensuite, un bref état de l'art est donné pour positionner le contexte de mes recherches. Enfin, l'ensemble de mes travaux dans ce domaine est présenté, à savoir : l'établissement de conditions de convergence vers un point fixe donné, la conception et la mise en œuvre d'un algorithme de détermination de la convergence à partir d'un état initial donné et l'étude du mixage synchronisme/asynchronisme pour stabiliser le comportement de ces systèmes tout en conservant les avantages de l'asynchronisme.

Formalisme des systèmes dynamiques

De manière générale, les systèmes dynamiques présentent un intérêt important pour la résolution de problèmes scientifiques complexes. Le plus souvent, ils s'écrivent sous la forme d'algorithmes itératifs dont la convergence donne la solution au problème traité. Ces algorithmes sont très largement utilisés dans un cadre séquentiel sur stations de travail. Jusqu'à ces dernières années, ils étaient également utilisés dans un cadre parallèle mais exclusivement en mode synchrone, ce qui restreignait leur utilisation au mieux aux grappes locales de calcul. Les travaux que j'ai menés dans ce domaine portent sur l'étude de l'utilisation de l'asynchronisme dans ces systèmes. Pour bien comprendre les travaux que j'ai effectués dans ce domaine, il est nécessaire de connaître les formalismes de base qui s'y rapportent. Ainsi, le but de ce chapitre est de présenter le formalisme général des systèmes dynamiques commun aux cas continu et discret.

Un système dynamique est défini par un ensemble de n éléments x_i , $i \in \{1, \dots, n\}$ à valeurs dans un ensemble E_i . L'ensemble des valeurs, ou états, des éléments x_i définit un vecteur x correspondant à l'état global du système :

$$x = (x_1, \dots, x_n) \in E = \prod_{i=1}^n E_i \quad (1.1)$$

De plus, l'état de chaque élément x_i évolue au cours du temps discret (itérations) selon des fonctions f_i , $i \in \{1, \dots, n\}$ définies de E dans E_i . Cela implique que l'évolution du système global est donnée au cours du temps par une fonction f telle que :

$$f = (f_1, \dots, f_n) , \quad f : E \mapsto E \quad \text{avec} \quad f_i : E \mapsto E_i \quad \forall i \in \{1, \dots, n\} \quad (1.2)$$

Plusieurs dynamiques peuvent alors être définies selon la façon dont on effectue les mises à jour des éléments et la prise en compte ou non de délais de communication entre les éléments.

La partie 1.1 est consacrée aux systèmes dynamiques synchrones en mettant en évidence la relation directe entre système dynamique et algorithme itératif parallèle. De même, la parallélisation de ces systèmes est décrite dans cette partie. Ensuite, leur extension aux systèmes dynamiques asynchrones est présentée en partie 1.2.

1.1 Dynamiques synchrones

La dynamique la plus simple est la dynamique *séquentielle* dans laquelle les éléments sont mis à jour un par un, les uns après les autres. Cette dynamique se rapproche du *mode série* décrit par François Robert dans [125].

À l'inverse, la dynamique *synchrone*, appelée aussi *mode parallèle* correspond à une mise à jour de tous les éléments de manière simultanée à chaque itération. Dans ce cas, l'équation d'évolution du système a la forme suivante :

$$x^{t+1} \equiv (x_1^{t+1}, \dots, x_n^{t+1}) = f(x^t) \equiv (f_1(x^t), \dots, f_n(x^t)) , \quad t = 0, 1, \dots \quad \text{avec } x^0 \text{ donné} \quad (1.3)$$

où

$$x_j^{t+1} = f_j(x^t) \equiv f_j(x_1^t, \dots, x_n^t) , \quad j = 1, \dots, n \quad (1.4)$$

Mais elle peut aussi s'exprimer sous la forme d'un algorithme itératif :

Algorithme 1 Algorithme itératif

Étant donné l'état initial $x^0 = (x_1^0, \dots, x_n^0)$

pour $t = 0, 1, \dots$ **faire**

pour $i = 1, \dots, n$ **faire**

$x_i^{t+1} = f_i(x^t)$

fin pour

fin pour

Enfin, il existe une dynamique intermédiaire entre ces deux premières qui est la dynamique *séquentielle par blocs*, appelée aussi *mode série-parallèle*. Dans ce mode, les éléments sont répartis dans des blocs fixés à l'avance qui sont mis à jour les uns après les autres. Par contre, les éléments d'un même bloc sont mis à jour en parallèle. Les modes séquentiel et parallèle sont des cas particuliers de ce mode là.

Quelle que soit la dynamique utilisée, lorsque le système atteint un état x^* tel que $f(x^*) = x^*$ alors il reste indéfiniment dans cet état et l'on dit que le système converge vers le point fixe x^* . En pratique, l'état x^* correspond à la solution du problème traité.

L'Algorithme 1 peut être exécuté de manière séquentielle sur un seul processeur contenant tous les éléments x_i ou de manière parallèle synchrone à grain fin en répartissant un élément x_i par processeur. Or, le nombre n d'éléments du système étant généralement très grand, il est rarement possible d'assigner un seul élément x_i à un processeur. Ainsi, une adaptation de cet algorithme est nécessaire et elle est réalisée par une décomposition par blocs du système.

Si l'on partitionne le vecteur x en m blocs $X_j, j \in \{1, \dots, m\}$ comportant chacun n_j éléments de x , et que l'on décompose la fonction f de manière similaire en m sous-fonctions $F_j, j \in \{1, \dots, m\}$, chaque F_j correspondant à la collection des fonctions d'évolution f_i des

éléments contenus dans le bloc j , les équations 1.3 et 1.4 peuvent être reformulées de la façon suivante :

$$X^{t+1} \equiv (X_1^{t+1}, \dots, X_m^{t+1}) = F(X^t) \equiv (F_1(X^t), \dots, F_m(X^t)) , \quad t = 0, 1, \dots \text{ avec } X^0 \text{ donné} \quad (1.5)$$

où

$$X_j^{t+1} = F_j(X^t) \equiv F_j(X_1^t, \dots, X_m^t) , \quad j = 1, \dots, m \quad (1.6)$$

Ainsi, nous obtenons une décomposition par blocs du système qui permet de répartir l'ensemble des n éléments x_i sur l'ensemble des m processeurs disponibles. L'algorithme 2 décrit la version itérative parallèle qui en découle.

Algorithme 2 Algorithme itératif parallèle

Étant donné l'état initial $X^0 = (X_1^0, \dots, X_m^0)$

pour $t = 0, 1, \dots$ **faire**

pour $j = 1, \dots, m$ **faire**

$X_j^{t+1} = F_j(X^t)$

fin pour

fin pour

Il est important de remarquer que la décomposition par blocs ne modifie pas la nature du système. En effet, comme les X_j sont des parties de x , chacun d'eux est à valeurs dans un produit cartésien d'ensembles E_i . Ainsi, les ensembles E'_j contenant les états des blocs X_j sont de même nature que les E_i qu'ils contiennent. Et comme ces X_j forment une partition de x , les états du système sont à valeurs dans $E' = \prod_{j=1}^m E'_j$. Le système obtenu peut donc être vu comme un système non décomposé à m éléments, de même nature que le précédent, à valeurs dans E' et dont la fonction d'activation est F . De plus, il y a équivalence directe entre les points fixes de ces deux systèmes puisque le dernier utilise intrinsèquement les espaces d'états et les fonctions d'évolution du premier.

Dans l'Algorithme 2, le synchronisme apparaît au niveau de la mise à jour du bloc j qui dépend uniquement de l'état global précédent. En effet, pour mettre à jour le bloc j , le $j^{\text{ème}}$ processeur doit avoir les valeurs à l'itération précédente de tous les éléments dont dépend F_j . Il est donc nécessaire d'utiliser des communications synchrones pour échanger ces valeurs entre les processeurs.

Du point de vue de l'évolution globale du système, les conditions de mise à jour des éléments étant les mêmes dans l'algorithme parallèle synchrone que dans l'algorithme séquentiel, on obtient exactement la même séquence d'états globaux du système au cours des itérations. Ainsi, le comportement global de l'algorithme parallèle synchrone est équivalent à celui de l'algorithme séquentiel et les conditions de convergence sont donc elles aussi identiques entre ces deux algorithmes. Ainsi, si un algorithme itératif séquentiel converge en k itérations, sa version parallèle synchrone convergera aussi en k itérations.

Concernant ces conditions de convergence, étant donné qu'elles diffèrent entre le cadre continu et le cadre discret, elles seront abordées séparément au cours des chapitres suivants.

1.2 Dynamiques asynchrones

L'asynchronisme n'ayant de sens que dans un contexte de calcul parallèle, nous reprenons dans cette partie la description du système par blocs. Il est rappelé que cela n'entraîne aucune perte de généralité puisque cette description par blocs englobe la description par éléments (en prenant un élément par bloc).

La notion d'asynchronisme la plus simple est une extension du cas séquentiel dans laquelle le seul élément mis à jour à chaque itération est choisi au hasard [99]. La notion plus générale est souvent appelée la dynamique *distribuée* ou *mode chaotique* et correspond à la mise à jour en parallèle d'un sous-ensemble non vide des éléments du système choisi aléatoirement [76]. Cependant, la condition de *fair sampling* est appliquée à cette dynamique pour en assurer une évolution représentative. Elle implique que les ensembles aléatoires doivent être choisis de façon à ce que chaque élément se mette à jour infiniment souvent sur l'ensemble des itérations. On peut noter que les dynamiques séquentielle, séquentielle par blocs et synchrone sont des cas particuliers de celle-ci. La dynamique fractionnaire proposée par Ballard [58] en est aussi un cas particulier puisque dans cette dernière, la taille des sous-ensembles est fixée. Enfin, les modes les plus complexes sont ceux qui incluent des retards (pouvant venir des communications ou des vitesses différentes des processeurs) entre les éléments d'un système chaotique. Il y en a deux possibles selon que les retards sont bornés ou non. Dans la communauté, le terme *asynchrone* est généralement réservé à ces deux modes et la distinction est faite par le terme *totalelement asynchrone* pour désigner la version à retards non bornés [59]. C'est sur ce dernier mode que mes travaux ont principalement porté et c'est donc celui-ci qui est détaillé dans la suite de cette partie. Pour éviter les surcharges, dans la suite de ce document, je ferai allusion à ce mode en utilisant simplement le terme *asynchrone*, sauf lorsque la distinction sera nécessaire.

Au niveau du formalisme, l'ajout de l'asynchronisme total dans le système présenté précédemment implique deux nouvelles notions qui sont complémentaires.

La première est ce que l'on appelle la stratégie de mise à jour, dénotée $J(t)$, qui indique pour chaque itération, l'ensemble des blocs $X_i, i \in \{1, \dots, m\}$ effectivement mis à jour. Ainsi, si à l'instant t , $k \in J(t)$ alors X_k est mis à jour en utilisant sa fonction d'évolution F_k . Par contre, si k n'est pas dans $J(t)$, alors X_k garde la valeur qu'il avait précédemment. Cette notion vient initialement des systèmes chaotiques dans lesquels le chaos est généré par cette même stratégie $J(t)$ qui est une séquence de sous-ensembles d'entiers de $\{1, \dots, m\}$. Cette notion permet notamment de simuler la différence de puissance de calcul entre les différents éléments du système.

La seconde notion est celle des retards entre les nœuds de calcul et donc entre les blocs. Comme mentionné précédemment, elle est propre aux systèmes asynchrones et permet de représenter les délais de communication entre les nœuds. Ces retards ont une grande importance car lors de la mise à jour d'un bloc, le processeur qui en a la charge n'attend pas d'avoir reçu les valeurs au temps précédent des autres blocs dont il dépend mais effectue sa mise

à jour avec les valeurs qu'il possède à cet instant. De plus, ces retards entre blocs peuvent eux-mêmes évoluer au cours du temps de façon à représenter les fluctuations de latence et de débit des liens de communication. Ainsi, si l'on définit le retard à l'instant t du bloc j par rapport au bloc i par $r_j^i(t)$, alors la valeur du bloc j utilisée pour mettre à jour le bloc i à l'instant t est celle à l'itération $s_j^i(t) = t - r_j^i(t)$.

Enfin, dans un système décomposé par blocs, l'asynchronisme s'applique entre les blocs mais les éléments à l'intérieur d'un même bloc restent synchronisés entre eux.

Finalement, l'équation d'évolution du système s'écrit pour chaque bloc $i \in \{1, \dots, m\}$:

$$\begin{aligned} X_i^{t+1} &= X_i^t & \text{si } i \notin J(t) \\ X_i^{t+1} &= F_i(X_1^{s_1^i(t)}, \dots, X_m^{s_m^i(t)}) & \text{si } i \in J(t) \end{aligned} \quad (1.7)$$

Et l'évolution de l'élément $j \in \{1, \dots, n_i\}$ du bloc i , noté $X_{i,j}$, s'écrit :

$$\begin{aligned} X_{i,j}^{t+1} &= X_{i,j}^t & \text{si } i \notin J(t) \\ X_{i,j}^{t+1} &= F_{i,j}(X_{1,1}^{s_1^i(t)}, \dots, X_{1,n_1}^{s_1^i(t)}, \dots, X_{i,1}^t, \dots, X_{i,n_i}^t, \dots, X_{m,1}^{s_m^i(t)}, \dots, X_{m,n_m}^{s_m^i(t)}) & \text{si } i \in J(t) \end{aligned} \quad (1.8)$$

On peut constater que la mise à jour ou non d'un élément ne dépend que du numéro de bloc dans lequel il se trouve. Cela signifie que tous les éléments d'un même bloc se mettent à jour aux mêmes instants. On a donc bien une exécution synchrone à l'intérieur des blocs. De plus, il n'y a pas de retards entre les éléments d'un même bloc et leurs valeurs à l'instant précédent sont utilisées lors des mises à jour. Enfin, on remarque que les retards sont identiques pour tous les éléments d'un même autre bloc que celui de l'élément considéré.

En reprenant la notation simplifiée par blocs, cela mène à l'Algorithme 3.

Algorithme 3 Algorithme itératif asynchrone

Étant donné un état initial $X^0 = (X_1^0, \dots, X_m^0)$

pour $t = 0, 1, \dots$ **faire**

pour $i = 1, \dots, m$ **faire**

si $i \in J(t)$ **alors**

$$X_i^{t+1} = F_i(X_1^{s_1^i(t)}, \dots, X_m^{s_m^i(t)})$$

sinon

$$X_i^{t+1} = X_i^t$$

fin si

fin pour

fin pour

On voit ici que l'on a toujours recours à une horloge globale pour décrire l'évolution du système. Cependant, il est important de comprendre que cette horloge est fictive et n'est utilisée que pour simplifier le formalisme et permettre une étude globale du système. Elle n'implique en rien une quelconque synchronisation des éléments entre eux ou avec un élément central.

Pour assurer une évolution significative du système, plusieurs règles sont appliquées :

- (a) $0 \leq r_j^i(t) \leq t$
- (b) $\forall i, j \in \{1, \dots, m\}, \lim_{t \rightarrow \infty} s_j^i(t) = \infty$, i.e. bien que les retards associés au bloc i ne soient pas limités, ils suivent l'évolution du système.
- (c) $\forall i \in \{1, \dots, m\}, |\{t, i \in J(t)\}| = \infty$, i.e. aucun élément n'est négligé par la règle de mise à jour (fair sampling).

La règle (a) est une condition évidente sur les bornes des retards. Par contre, la règle (b) est plus subtile et permet d'assurer que la même version d'un élément ne peut pas être indéfiniment utilisée pour les mises à jour des autres éléments. En effet, il est par exemple possible d'avoir des $s_j^i(t)$ dans l'intervalle $[\frac{t}{2}, t]$ mais pas dans l'intervalle $[0, t]$. Enfin, la règle (c) assure que tous les éléments sont régulièrement mis à jour.

Il est intéressant de noter que ce modèle est le plus général des algorithmes itératifs parallèles. Du point de vue de son évolution globale, on peut aisément constater que les conditions de mise à jour des blocs et donc des éléments ne sont plus les mêmes que dans les algorithmes séquentiel et synchrone. Ainsi, la séquence des états globaux du système suivie au cours des itérations peut être différente et le comportement global de l'algorithme asynchrone n'est plus équivalent aux deux algorithmes précédents. Une étude particulière des conditions de convergence dans le contexte asynchrone est donc nécessaire. Néanmoins, le fait que le cas asynchrone soit plus général que les cas synchrone et séquentiel, qui sont en fait des cas particuliers d'asynchronisme, implique que la convergence en asynchrone assure la convergence des versions synchrone et séquentielle. Dans ce cas, la convergence de l'algorithme asynchrone demande généralement plus d'itérations que celle de la version synchrone/séquentielle. Par contre, l'asynchronisme ne modifie pas le nombre et la valeur des points fixes du système dynamique, ce qui est très important pour l'intérêt pratique de ces algorithmes.

Les formalismes des systèmes dynamiques synchrone et asynchrone présentés précédemment permettent de mettre en évidence les différences de fonctionnement entre les versions synchrone et asynchrone. Ces formalismes sont très utiles pour l'étude théorique du comportement de ces systèmes, notamment pour prouver leur convergence, mais aussi d'autres propriétés particulières telles que la présence de cycles.

Dans les chapitres suivants, les spécificités liées au cadre d'utilisation, continu ou discret, seront précisées et leurs conséquences sur le comportement des systèmes dynamiques seront mises en évidence.

Systèmes dynamiques continus asynchrones

Les systèmes dynamiques continus sont les plus souvent utilisés en pratique car la plupart des problèmes scientifiques sont définis dans des domaines continus. Diverses études ont montré que ces algorithmes sont applicables à la résolution d'équations différentielles [126, 117, 89, 133], de systèmes markoviens [109, 136, 129] ou encore de problèmes d'optimisation [63, 64, 95]. Nous pourrions voir, tout au long de ce chapitre, que l'asynchronisme apporte une souplesse plus grande à ces algorithmes et permet ainsi leur utilisation dans des contextes de calcul plus performants mais aussi plus contraignants que sont les méta-grappes de calcul. Nous verrons également que certaines techniques classiques en parallélisme doivent être adaptées pour conserver cette souplesse et assurer les bonnes performances escomptées.

La première partie de ce chapitre présente les particularités du contexte continu sur le formalisme général donné au Chapitre 1. La partie 2.2 a pour objectif de replacer ces systèmes dans le contexte plus général des algorithmes itératifs parallèles et de présenter une classification de ces algorithmes. La partie 2.3 met en évidence l'intérêt général de l'asynchronisme dans les systèmes dynamiques continus sur grappes de machines et méta-grappes. L'adaptation à l'asynchronisme des techniques classiques en parallélisme et en calcul itératif que sont l'équilibrage de charge et la détection de la convergence est respectivement présentée dans les parties 2.4 et 2.5. Enfin, la partie 2.6 présente une comparaison de différents environnements de développements permettant la mise en œuvre des algorithmes asynchrones et dégage les caractéristiques nécessaires à un tel environnement pour assurer l'efficacité de ces algorithmes.

2.1 Formalisme spécifique

La spécificité du cadre continu par rapport au formalisme général porte sur la nature des ensembles E_i du système non décomposé qui sont des sous-ensembles de \mathbb{R} ou \mathbb{R} lui-même, ce qui donne dans une version décomposée par blocs des ensembles E'_i dans \mathbb{R}^{n_i} où n_i est le nombre d'éléments dans le bloc i .

Si cela ne change pas les équations d'évolution et algorithmes vus au chapitre précédent, cela revêt par contre une importance considérable au niveau du comportement du système et

donc au niveau de sa convergence. En effet, selon le type de problème traité par le système dynamique, les conditions de convergence ne s'expriment pas de la même façon. Il est donc nécessaire de faire une distinction entre les problèmes linéaires et les problèmes non linéaires.

2.1.1 Convergence pour les problèmes linéaires

Dans le cadre de problèmes linéaires, l'algorithme itératif s'écrit généralement sous la forme :

$$x^{t+1} = M.x^t + G.b \quad (2.1)$$

où M et G sont des matrices définies en fonction des données initiales et de la méthode de calcul utilisée (Jacobi ou Gauss-Seidel). M est appelée la matrice d'itération et ce processus n'a de sens que si $I - M$ est inversible (I est la matrice identité).

Il est montré que l'algorithme itératif synchrone ainsi défini converge quel que soit l'état initial x^0 si est seulement si $\rho(M) < 1$. Par contre, les travaux [115, 59, 135] montrent que la version asynchrone de cet algorithme itératif converge si $\rho(|M|) < 1$ où $|M|$ est la valeur absolue élément par élément de la matrice M . Par la suite, D.P.Bertsekas et J.N.Tsitsiklis [67] ont montré que cette condition n'est pas seulement suffisante mais est nécessaire.

Dans ce contexte, on voit donc que l'asynchronisme est un peu plus contraignant sur les conditions de convergence.

2.1.2 Convergence pour les problèmes non linéaires

Dans le cas de problèmes non linéaires, les conditions de convergence sont exprimées en fonction d'une norme sur l'ensemble des états globaux du système. Dans le cadre des systèmes dynamiques, l'ensemble des états globaux E est un produit cartésien et la norme qui doit être utilisée est alors la norme de type max définie par :

$$\|x\| = \max_i \frac{\|x_i\|_i}{\gamma_i} \quad (2.2)$$

où $\|\cdot\|_i$ est la norme sur l'ensemble d'états de l'élément i et $\gamma_i > 0$. Avec une telle norme, on peut définir une *contraction* par :

$$\|f(x) - f(y)\| \leq \alpha \|x - y\|, \quad \forall x, y \in E \text{ et } 0 < \alpha < 1 \quad (2.3)$$

ou encore une *pseudo-contraction* (contraction par rapport à x^*) par :

$$\|f(x) - x^*\| \leq \alpha \|x - x^*\|, \quad \forall x \in E, \quad x^* \in E \text{ tel que } f(x^*) = x^* \text{ et } 0 < \alpha < 1 \quad (2.4)$$

Il a été démontré par M.N.El Tarazi [135] que ces deux contextes impliquent un point fixe x^* unique et une convergence du système vers ce point fixe quel que soit son état initial.

De leur côté, Bertsekas et Tsitsiklis [67] ont montré que la convergence vers un point fixe de f est assurée en mode asynchrone si, en plus de la condition de convergence synchrone, une autre condition, communément appelée condition des boîtes (*box condition* en anglais), est vérifiée. Globalement, cela revient à dire que non seulement, l'ensemble des états globaux possibles se réduit au cours du temps jusqu'à ne plus contenir qu'un seul élément qui est un point fixe de f mais qu'en plus, cette réduction des possibilités se fait par une séquence de sous-ensembles emboîtés compacts. Si l'on considère E^t l'ensemble des états globaux possibles à l'instant t , alors cet ensemble peut s'écrire sous la forme $E_1^t \times E_2^t \times \dots \times E_n^t$ où E_i^t est l'ensemble des valeurs possibles sur l'élément i à t .

Là aussi, on voit donc qu'il y a une contrainte supplémentaire impliquée par l'asynchronisme pour assurer la convergence. Cependant, celle-ci n'est pas aussi restrictive que précédemment et les algorithmes asynchrones restent applicables à un grand nombre de problèmes scientifiques.

2.2 État de l'art

Comme il a été mentionné en introduction, les algorithmes itératifs sont des méthodes de résolution de problèmes fonctionnant par approximations successives. Ces méthodes sont connues depuis très longtemps puisque la première mention connue en est faite par Gauss pour la résolution d'un système d'équations de dimension quatre. Mais c'est à partir des années cinquante que la théorie autour de ces méthodes a vraiment été développée avec les travaux de D.M.Young [142, 143] et la mise au point de la méthode du gradient conjugué [106, 98].

Un peu plus tard, l'apparition de systèmes de calcul plus complexes a conduit les chercheurs à s'intéresser aux algorithmes itératifs parallèles [82, 77]. C'est à cette époque que le modèle général correspondant aux algorithmes itératifs asynchrones a été initié par D.Chazan et W.Miranker [76] sous la forme des méthodes chaotiques de point fixe et généralisé aux itérations asynchrones par G.M.Baudet [59]. Le problème qui s'est vite posé avec ces algorithmes fut l'analyse de leur convergence. Les travaux initiaux de J-C.Miellou sur la contraction vectorielle [115] ont été repris et étendus pour mener à deux résultats théoriques majeurs, le théorème de M.N.El Tarazi [135] et celui de Bertsekas et Tsitsiklis [67]. Le premier utilise les notions de contraction alors que le second est fondé sur la notion d'ensembles emboîtés. Ces notions ont été abordées au paragraphe précédent.

Ces algorithmes ont été assez largement étudiés au niveau théorique (on peut encore citer [65, 68, 55, 84]). Certains aspects bien maîtrisés dans les algorithmes synchrones ont dû être reconsidérés dans le cadre asynchrone, comme par exemple la méthode et le critère utilisés pour la détection de convergence ou encore la procédure d'arrêt. Ces problèmes liés à la terminaison du processus ont fait l'objet de plusieurs études [66, 128, 60, 71], mais dans des contextes ou avec des conditions assez restrictives.

Enfin, malgré un nombre important d'études théoriques, les algorithmes itératifs asynchrones ont été assez peu mis en œuvre, peut-être par manque d'environnements de programmation adaptés. En effet, jusqu'à ces dernières années, la plupart des environnements de programmation parallèle, tels que PVM [90] et MPI [94], étaient principalement orientés vers les algorithmes synchrones et ne permettaient pas une mise en œuvre efficace des algorithmes asynchrones. Il manquait notamment les mécanismes permettant de gérer efficacement les communications asynchrones. L'apparition récente d'environnements de développement multiprocesseurs supprime cette barrière et ouvre enfin la voie à des mises en œuvre à la hauteur des attentes théoriques.

C'est dans ce contexte que j'ai orienté mes travaux de recherche sur la conception et la mise en œuvre d'algorithmes itératifs asynchrones pour la résolution de problèmes scientifiques. La classification proposée ci-dessous permet de décrire les grands types d'algorithmes itératifs parallèles existant avec leurs éventuelles variantes et de préciser celui qui fait l'objet de mes travaux.

2.2.1 Classification des algorithmes itératifs parallèles

La classe des algorithmes itératifs parallèles étant relativement importante, cette dénomination seule ne suffit pas à pouvoir identifier précisément un algorithme particulier de cette classe, notamment en termes de schéma de mise en œuvre et de comportement global. Les différences significatives provenant essentiellement de la nature synchrone ou asynchrone des itérations et des communications, une classification a été définie en fonction de ces critères.

Elle débouche sur trois sous-classes d'algorithmes itératifs parallèles qui sont présentés dans les paragraphes suivants.

Itérations Synchrones - Communications Synchrones (SISC)

Cette catégorie correspond au schéma le plus communément utilisé. À chaque itération, chaque processeur attend d'avoir reçu les valeurs provenant des autres processeurs calculées à l'itération précédente avant de commencer l'itération suivante. Les échanges de données sont donc effectués à la fin de chaque itération par des communications synchrones globales. Une représentation schématique du flux d'exécution d'un tel algorithme est donnée dans la Figure 2.1 dans le cas de deux processeurs.

Comme évoqué dans la présentation des formalismes au Chapitre 1, la séquence des itérations est, dans ce cas, identique à celle du cadre séquentiel. Ainsi, les conditions de convergence de ces algorithmes sont les mêmes que celles des algorithmes itératifs séquentiels. C'est une des raisons du succès de ce type d'algorithmes puisqu'il ne nécessite aucune étude de convergence supplémentaire et permet en outre une parallélisation relativement directe d'un algorithme itératif séquentiel donné.

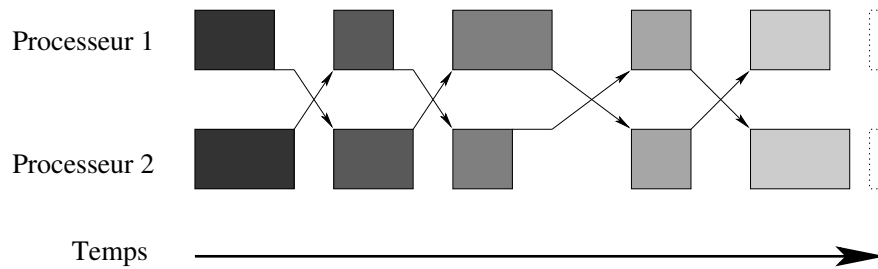


FIG. 2.1 – Schéma d'exécution d'un algorithme SISC avec deux processeurs. Les blocs grisés représentent les itérations et les flèches représentent les communications de données. Les parties horizontales des flèches indiquent les mises en attentes des communications dues aux synchronisations.

Cependant, les communications synchrones pénalisent fortement l'efficacité de ces algorithmes. En effet, comme on peut le voir sur la Figure 2.1, il peut y avoir de nombreux temps morts (représentés par des espaces blancs) entre les itérations (blocs gris) selon la rapidité du réseau utilisé. Si ces synchronisations n'ont qu'un effet relativement réduit dans des contextes de calcul où le réseau d'interconnexion est très rapide, tels que les machines parallèles ou même les grappes locales, il en est tout autrement dans les contextes de méta-grappes où les liens de communications entre machines peuvent être de qualités très différentes. Dans ce cas, la perte de performance est généralement si importante qu'il n'est pas intéressant d'utiliser cet algorithme dans ce contexte.

Itérations Synchrones - Communications Asynchrones (SIAC)

Cette catégorie a été développée dans le but principal de résoudre les problèmes de performance liés aux algorithmes SISC. Le principe est de conserver un schéma itératif synchronisé pour avoir le même comportement global et donc les mêmes conditions de convergence que précédemment, tout en effectuant les échanges de données entre processeurs de manière asynchrone de façon à réaliser un recouvrement des calculs par des communications.

En fait, avant de commencer une itération, chaque processeur attend toujours d'avoir reçu toutes les données calculées à l'itération précédente nécessaires à son calcul. Par contre, la différence se fait au niveau de l'envoi de ces données. En effet, chaque donnée (ou groupe de données) est envoyée de manière asynchrone aux processeurs qui en ont besoin. En général, ce schéma est associé à une flexibilité des communications qui consiste à envoyer les données ou groupes de données, non pas à la fin de l'itération courante, mais dès que ces données ont été mises à jour, pendant l'itération. Ce schéma compte sur le fait que les données ainsi envoyées à un autre processeur ont une bonne probabilité d'arriver sur celui-ci avant qu'il ait terminé l'itération courante. Ainsi, elles seront déjà disponibles sur ce processeur lorsqu'il devra commencer l'itération suivante et il n'aura donc pas à attendre leur réception. Cela revient donc à faire un recouvrement partiel des calculs par des communications. Il est partiel car le calcul de la première donnée ou du premier groupe de données ne peut être recouvert par

des communications. Un exemple de flux d'exécution de ce type d'algorithmes est donné dans la Figure 2.2 pour deux processeurs. On peut voir notamment que l'ordre des communications peut ne pas être respecté.

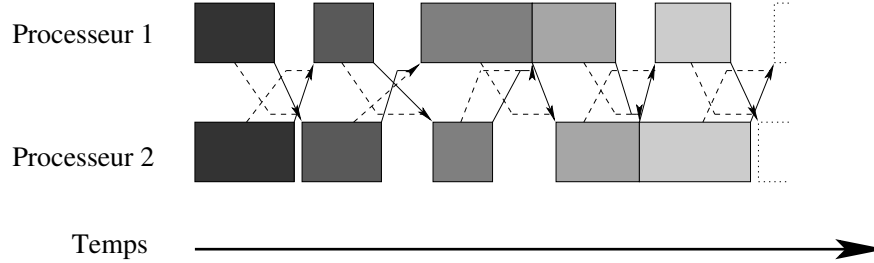


FIG. 2.2 – Schéma d'exécution d'un algorithme SIAC avec deux processeurs et des envois de données en deux groupes. Les flèches discontinues représentent les envois du premier groupe de données et les flèches continues, celles du second groupe. Les parties horizontales des flèches indiquent les attentes d'incorporation dans les calculs des données reçues.

Du point de vue du comportement global, nous restons dans le même cadre que les algorithmes séquentiels. Étant donné que chaque processeur commence l'itération suivante dès qu'il a reçu toutes les données mises à jour dont il a besoin, les processeurs peuvent ne pas commencer cette itération en même temps. Néanmoins, pour cette même raison, un processeur qui aurait fini l'itération courante avant tous les autres ne pourrait pas commencer l'itération suivante avant eux. Il est donc impossible d'avoir à un instant donné des processeurs qui calculent des itérations de numéros différents. En fait, à chaque instant, les processeurs sont soit en train de calculer la même itération (même numéro) soit en train d'attendre les données pour l'itération suivante. La notion de synchronisme des itérations reste donc valable dans ce schéma et les conditions de convergence sont donc identiques aux algorithmes SISC et séquentiels.

Le recouvrement des calculs par des communications a pour effet de réduire les temps morts entre les itérations et donc d'améliorer les performances globales. Cependant, comme on peut le voir sur la Figure 2.2, il ne supprime pas complètement ces temps morts. En fait, la communication des données vers un processeur peut être plus longue que le calcul de l'itération courante sur ce processeur. Il devra donc attendre, une fois l'itération courante terminée, d'avoir reçu toutes les données pour passer à l'itération suivante. De plus, l'envoi des dernières données calculées sur le processeur le plus en retard lors d'une itération ne peut pas être recouvert par des calculs.

Itérations Asynchrones - Communications Asynchrones (AIAC)

Nous avons vu dans les catégories précédentes que la synchronisation des itérations empêche d'obtenir un recouvrement complet des calculs par les communications. Ainsi, pour obtenir un tel recouvrement, il est nécessaire de désynchroniser non seulement les communications, mais aussi les itérations.

Dans ce contexte, tous les processeurs effectuent leurs itérations sans tenir compte de la progression des autres processeurs. Ils n'attendent donc pas de recevoir les données calculées à l'itération précédente mais continue leurs calculs en utilisant les versions des données qu'ils possèdent à cet instant. Comme les processeurs n'attendent plus les données provenant des autres processeurs, il n'y a plus de temps morts entre les itérations, comme on peut le voir sur la Figure 2.3. Par contre, les itérations sont désynchronisées, ce qui implique que les processeurs peuvent exécuter des itérations différentes au même instant. C'est pour cette raison que l'évolution locale, et donc aussi l'évolution globale, est différente des cas précédents. Le schéma de la Figure 2.3 correspond au modèle de base de Bertsekas [67] ou El Tarazi [135].

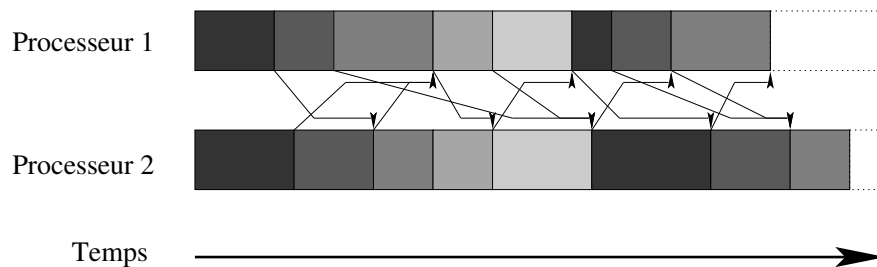


FIG. 2.3 – Schéma d'exécution d'un algorithme AIAC avec deux processeurs. Les parties horizontales des flèches indiquent les attentes d'incorporation dans les calculs des données reçues.

Nous avons vu au chapitre précédent que les algorithmes asynchrones demandent généralement plus d'itérations pour converger que leurs homologues synchrones ou séquentiels. On pourrait donc se dire qu'ils sont a priori plus lents que ceux-ci, ce qui est vrai en termes d'itérations. Cependant, les pénalisations dues aux communications synchrones sont généralement si importantes qu'elles dépassent le temps mis pour effectuer les itérations supplémentaires de la version asynchrone. Ainsi, selon le contexte de calcul, il est possible d'avoir un algorithme itératif qui soit plus rapide qu'un autre tout en effectuant plus d'itérations. Les contextes propices à ce genre de phénomène sont justement ceux des méta-grappes de calcul dans lesquels les liens de communications sont souvent hétérogènes avec certains liens bien plus lents que d'autres. L'efficacité des algorithmes asynchrones dans ce contexte vient du fait qu'ils sont moins sensibles aux délais de communication et à leurs variations que les algorithmes synchrones. Ils présentent même une certaine tolérance aux pertes des messages de données puisqu'une telle perte ne perturbe ni l'expéditeur, ni le récepteur du message. Tous deux continuent leurs calculs et un autre message de données est envoyé lors d'une itération ultérieure. Cependant, cette tolérance ne s'applique pas aux messages de contrôle du processus utilisés pour la détection de la convergence globale et l'arrêt du système. De plus, pour obtenir des algorithmes asynchrones plus efficaces que leurs homologues synchrones dans plus de contextes de calcul, il est essentiel de minimiser le nombre d'itérations supplémentaires induites par l'asynchronisme.

C'est dans ce but que plusieurs variantes du modèle asynchrone initial ont été mises au point. Ces variantes jouent sur le moment où sont faites les communications ou encore sur

le moment où les données reçues sont incorporées dans le calcul. Parmi celles-ci, deux sont particulièrement intéressantes, il s'agit des algorithmes asynchrones avec communications semi-flexibles ou avec communications flexibles.

Dans le premier cas, les données sont toujours envoyées à la fin de l'itération. Par contre, elles sont incorporées dans les calculs dès leur réception sur le processeur. Le schéma d'exécution devient alors celui de la Figure 2.4.

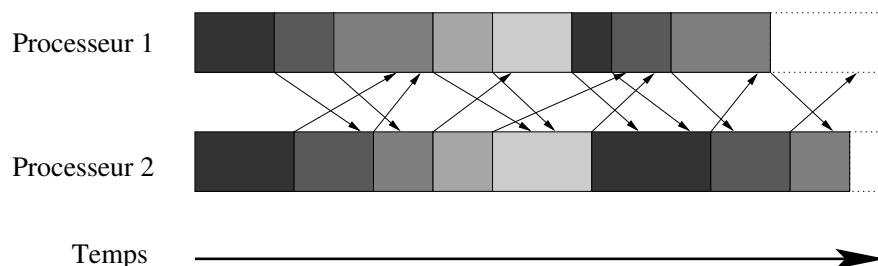


FIG. 2.4 – Schéma d'exécution d'un algorithme AIAC avec communications semi-flexibles sur les réceptions

L'objectif de cette variante est d'accélérer la convergence de l'algorithme en prenant en compte le plus vite possible les données les plus récentes. Néanmoins, cela n'est pas toujours possible selon le type de calcul effectué. Dans ce cas, une autre solution est de conserver la prise en compte des données reçues à la fin de l'itération mais d'utiliser le même schéma d'envoi des données que celui utilisé dans les algorithmes SIAC. Ainsi, les données sont découpées en groupes qui sont envoyés en cours d'itération, dès qu'ils ont été mis à jour. Les groupes peuvent être réduits à un élément mais, généralement, un regroupement est effectué pour éviter de surcharger le réseau avec un grand nombre de petits messages. Cette méthode a aussi pour effet d'accélérer la convergence en rendant les données mises à jour disponibles, le plus vite possible, aux autres processeurs. Le schéma d'exécution de cette autre variante semi-flexible est donné dans la Figure 2.5.

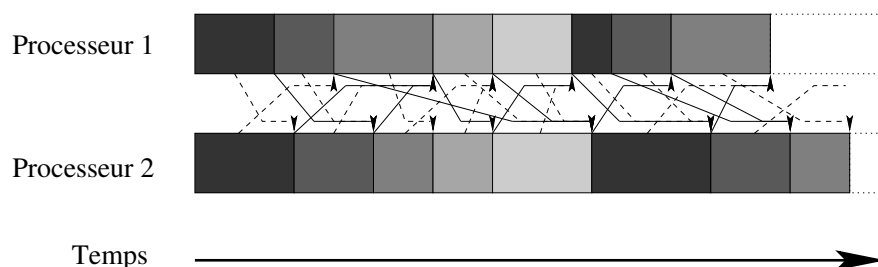


FIG. 2.5 – Schéma d'exécution d'un algorithme AIAC avec communications semi-flexibles sur les envois en deux groupes de données. Les flèches discontinues représentent les envois du premier groupe de données et les flèches continues, celles du second groupe. Les parties horizontales des flèches indiquent les attentes d'incorporation dans les calculs des données reçues.

La seconde variante, introduite par Miellou, D.El Baz et P.Spitéri dans [116] (voir aussi [62, 85]) revient à faire une combinaison des deux variantes semi-flexibles précédentes. En plus de l'incorporation immédiate des données reçues, les envois des données sont effectués par groupes dès leur mise à jour au cours de l'itération. La Figure 2.6 donne le schéma d'exécution qui en découle. Cette dernière solution permet de se rapprocher encore un peu plus de la vitesse de convergence en mode synchrone.

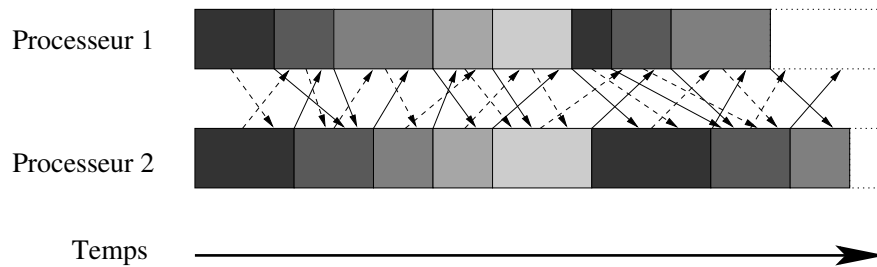


FIG. 2.6 – Schéma d'exécution d'un algorithme AIAC avec communications flexibles en deux groupes de données. Les flèches discontinues représentent les envois du premier groupe de données et les flèches continues, celles du second groupe.

Mes travaux de recherche présentés dans la suite portent sur les algorithmes AIAC en général. Selon les cas, l'une ou l'autre des variantes présentées ci-dessus peut être utilisée mais de nouvelles variantes sont aussi proposées pour répondre au mieux aux contraintes liées au contexte d'étude ou au problème traité.

2.3 Intérêt général de l'asynchronisme

Généralement, les personnes familières des algorithmes itératifs synchrones qui sont confrontées pour la première fois à la notion d'algorithmes itératifs asynchrones sont sceptiques quant aux meilleures performances de ces derniers. Cette tendance peut sans doute s'expliquer l'association légitime entre le nombre d'itérations nécessaire à la convergence et les performances globales de l'algorithme.

Les études [19, 5] que j'ai menées dans ce domaine tendent à montrer que cette idée n'est plus vraie dans les contextes de calcul parallèle récents et notamment les méta-grappes. L'intérêt de l'asynchronisme est mis en valeur au travers de deux exemples représentatifs des problèmes scientifiques habituellement rencontrés, un problème linéaire creux indépendant du temps et un problème non linéaire dépendant du temps. Dans chaque cas, l'algorithme asynchrone est détaillé puis une comparaison expérimentale est effectuée avec son homologue synchrone dans les contextes de grappes de calcul et/ou de méta-grappes.

2.3.1 Schéma algorithmique général

Que ce soit pour résoudre un problème linéaire ou un problème non linéaire, le schéma algorithmique est globalement identique. Cette partie présente donc ce schéma général utilisé pour résoudre les deux problèmes. Les éléments spécifiques à chaque problème sont donnés dans les parties respectives suivantes. Dans ce qui suit, nous considérons avoir à notre disposition un ensemble de N machines pouvant communiquer entre elles.

La plupart de problèmes scientifiques reviennent à calculer un ensemble de valeurs (vecteur, matrice...). Le parallélisme est alors classiquement exploité en répartissant ces données à calculer sur l'ensemble des processeurs. Cette répartition n'est généralement pas faite au hasard mais tient compte des dépendances entre les données à calculer de façon à minimiser les communications nécessaires entre les processeurs. Cela permet de minimiser les goulots d'étranglements et/ou les collisions dans le réseau d'interconnexion mais aussi de faire correspondre plus facilement l'organisation logique des processeurs à leur organisation physique. En effet, il n'est pas rare, surtout dans le contexte de méta-grappes, d'avoir des machines qui ne puissent communiquer entre elles directement et qui doivent passer par des processeurs particuliers. Ces communications par machines interposées doivent être évitées au maximum pour garantir de bonnes performances. Ainsi, les processeurs doivent être organisés de façon à minimiser les liens physiques de communication.

Une fois les données du problème convenablement réparties, le schéma itératif asynchrone général est décrit par l'Algorithme 4. Chaque processeur met à jour ses données locales en utilisant les valeurs locales obtenues à l'itération précédente et les dernières valeurs reçues des données nécessaires au calcul provenant des autres processeurs. Les données locales mises à jours sont ensuite envoyées de manière asynchrone aux processeurs qui en ont besoin. Comme nous le verrons en 2.6, une mise en œuvre à base de processus légers est recommandée pour tirer partie au maximum de l'asynchronisme. Ces processus ont l'avantage d'être indépendants du processus principal tout en partageant le même espace de données. C'est donc ce type de programmation qui est présenté ici. Ainsi, les envois de données sont effectués dans des processus légers indépendants du processus de calcul et activés lorsque nécessaire pour appeler une fonction d'envoi décrite dans l'Algorithme 5. Lorsque l'organisation logique des processeurs correspond à l'organisation physique, les envois sont limités aux processeurs voisins. Dans le cas contraire, il est souvent intéressant de construire sur chaque processeur une liste des dépendances sortantes, c'est-à-dire, une liste des processeurs qui ont besoin de données calculées localement ainsi que la liste des données concernées. La réception des données doit aussi se faire de manière asynchrone, si possible, par un processus léger scrutant l'arrivée des messages. Ainsi, lorsque des données arrivent, ce processus appelle une fonction de réception, donnée en Algorithme 6, qui les récupère et les place au bon endroit dans le tableau des données locales.

Pour effectuer ces échanges de données, n'importe laquelle des variantes d'AIAC énumérées dans la partie 2.2.1 peut être utilisée. Par exemple, selon que l'on utilise une version rigide ou flexible, les données reçues seront respectivement placées dans le tableau des nouvelles données

Algorithme 4 Algorithme itératif asynchroneAnciennesValeurs = *Tableau des données calculées localement à l'itération précédente*NouvellesValeurs = *Tableau des données calculées localement à l'itération courante*

Initialisation du système

Initialisation des données dans AnciennesValeurs

répéter

Calcul des nouvelles valeurs dans NouvellesValeurs en utilisant AnciennesValeurs

Envois asynchrones des données calculées aux processeurs qui en ont besoin

Copie de NouvellesValeurs dans AnciennesValeurs

Détection de la convergence globale

jusqu'à Convergence globale est détectée

Affichage ou sauvegarde des données locales

Arrêt du système

Algorithme 5 Fonction EnvoieDonnées(NumDest : entier)Récupération des données à envoyer au processeur destinataire *NumDest* dans le tableau des données locales

Envoi de ces données au processeur NumDest

ou dans celui des anciennes données. Cependant, pour éviter des surcharges du réseau et des situations incohérentes dues à l'utilisation simultanée de l'une des fonctions de communication (envoi ou réception) sur un même processeur, un système de mutex en conjonction avec des communications bloquantes a été utilisé. Comme les communications sont effectuées dans des processus légers indépendants du processus de calcul, leur nature bloquante ne synchronise pas les itérations du calcul. Les mutex sont activés sur un processeur au début de chaque type de communication (envoi ou réception) et empêchent l'utilisation de ce type de communication tant que celle en cours n'est pas terminée. À la fin de la communication, le mutex est désactivé pour autoriser à nouveau ce type de communication. Les réceptions de données pendant une itération ne posent pas de problème en elles-mêmes. Les communications étant bloquantes, les mutex sur les réceptions sont d'ailleurs surtout utiles pour assurer qu'il n'y a plus de réception en cours lors de la procédure d'arrêt. La difficulté au niveau des réceptions est plutôt d'assurer que chaque écriture des données reçues dans le tableau local des données soit atomique pour éviter les problèmes d'incohérence. Au niveau de la présentation des algorithmes, les mutex étant généralement inhérents à l'environnement de gestion des processus légers, ils sont considérés comme faisant partie intégrante des communications et ne sont pas indiqués explicitement. Cela permet en outre de conserver une présentation simple et claire

Algorithme 6 Fonction ReçoitDonnées()

Réception des données

Stockage des données reçues à la position qui leur correspond (selon l'expéditeur) dans le tableau des données locales

des algorithmes.

Par rapport à la classification des algorithmes AIAC, donnée en 2.2.1, le schéma algorithmique présenté ici est apparenté à la variante semi-flexible sur les réceptions. Cependant, dans notre cas, l'utilisation des mutex rend les communications exclusives, c'est-à-dire qu'il ne peut pas y avoir deux communications du même type au même moment sur un même processeur. Cela donne le schéma d'exécution présenté dans la Figure 2.7.

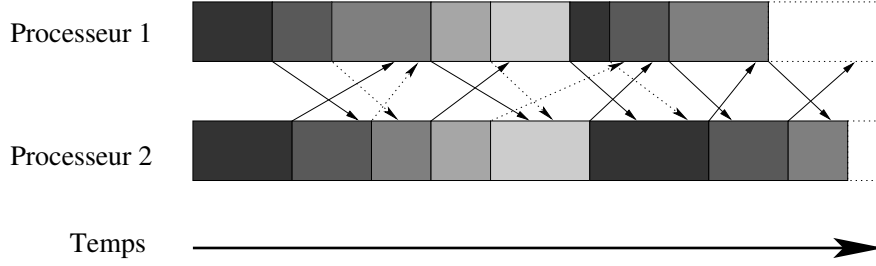


FIG. 2.7 – Schéma d'exécution d'un algorithme AIAC avec communications semi-flexibles sur les réceptions et communications exclusives. Les flèches en pointillés représentent les communications qui ne sont pas effectuées à cause de l'exclusivité des communications.

Finalement, le processus est répété jusqu'à la convergence du processus itératif. Celle-ci étant asymptotique dans le cadre continu, la solution exacte n'est généralement pas atteinte et le processus doit s'arrêter lorsque l'approximation est jugée suffisamment proche de la solution. Cette convergence est considérée atteinte lorsque le résidu entre deux itérations consécutives (k et $k - 1$) devient suffisamment petit par rapport à une norme donnée :

$$\text{norm}(x^k, x^{k-1}) < \epsilon, \quad k > 0 \quad \text{et} \quad \epsilon \in \mathbb{R}^+ \quad (2.5)$$

Généralement, la norme du max est utilisée et le critère d'arrêt du processus est donc :

$$\max_i |x_i^k - x_i^{k-1}| < \epsilon \quad (2.6)$$

Lorsque (2.6) est vérifiée, on dit que le processus a convergé à l'itération k .

L'algorithme présenté ici utilise la solution la plus simple qui consiste à rassembler l'état d'avancement de tous les processeurs sur un seul, qui sera appelé le maître. Une version décentralisée de détection de la convergence globale est présentée dans la partie 2.5. En pratique, les problèmes principaux qui se posent pour la détection de la convergence globale sont d'une part, la détection de la convergence locale et d'autre part, la connaissance de l'état global du système.

Concernant le premier point, pour que le processeur maître puisse détecter la convergence globale, il faut d'abord que chaque processeur détecte sa propre convergence locale. Comme indiqué ci-dessus, celle-ci est basée sur l'évolution du résidu entre deux itérations consécutives. Or, ce résidu ne suit généralement pas une décroissance monotone mais effectue des oscillations autour de cette décroissance. Ainsi, lorsqu'il atteint le seuil de précision souhaité, il peut

alterner entre des passages au-dessous et au-dessus du seuil. Ce n'est que lorsque la convergence locale est réellement atteinte qu'il restera indéfiniment sous le seuil. À l'inverse, avant la convergence locale, toutes les séquences d'itérations consécutives effectuées sous le seuil ont une longueur finie. En théorie, il suffit donc d'atteindre une séquence d'itérations sous le seuil dont la longueur est strictement supérieure à ces dernières pour détecter la convergence locale. Malheureusement, bien que ce nombre théorique d'itérations (noté **NbItCvLoc**) existe et soit fini lorsque le processus itératif converge, il n'est pas possible de l'évaluer de manière précise en pratique. En fait, il n'y a actuellement aucun moyen d'assurer une convergence locale définitive sur un processeur sans modifier le processus itératif, comme dans [69]. L'heuristique utilisée consiste donc à *supposer* que la convergence locale est atteinte lorsqu'un certain nombre d'itérations consécutives (noté **NbItCvLocSup** et fixé arbitrairement) ont été effectuées sous le seuil de convergence. Lorsque c'est le cas, l'état local du processeur passe en convergence locale supposée et cet état est envoyé au maître. Par contre, si le résidu repasse au-dessus du seuil, la convergence locale supposée est annulée et le nouvel état local est envoyé au maître pour notifier ce changement.

Concernant le second point, le processeur maître est choisi parmi les N processeurs participant au calcul. Il effectue donc des calculs comme les autres processeurs et a une petite charge supplémentaire qui consiste à recevoir les états d'avancement des autres processeurs et à détecter la convergence globale du système. Comme les réceptions sont effectuées de manière asynchrone et que la détection de convergence n'implique que très peu de calculs, on peut considérer que cette surcharge sur le processeur maître est négligeable par rapport aux calculs du processus itératif. Sur ce processeur maître, un tableau contenant l'état de tous les processeurs est mis à jour dès qu'un message d'état est reçu. La convergence globale est détectée lorsque tous les éléments du tableau indiquent une convergence locale. Dans ce cas, le maître envoie un message de convergence globale à tous les autres processeurs. La réception de ces messages est gérée sur les autres processeurs par une fonction spécifique qui consiste simplement à placer la valeur de **ConvGlobale** à Vrai. Cependant, si la convergence locale est annulée sur un processeur, il est nécessaire que le message d'état annulant cette convergence arrive sur le processeur maître avant que celui-ci ne sorte de la boucle du processus itératif, afin d'éviter une terminaison prématurée de celui-ci. Il faut donc mettre en place un mécanisme qui permette de continuer le processus itératif pendant un certain temps après la détection de convergence globale de façon à être sûr de ne pas rater une annulation éventuelle. Ce temps d'attente (noté **TpsMaxAnnul**) est défini par le temps maximal de transmission d'un message d'état d'un processeur vers le maître. Pour assurer le bon fonctionnement de cet algorithme, il est donc nécessaire de se placer dans un contexte où les retards sont bornés, c'est-à-dire dans lequel on a :

$$\exists B \in \mathbb{N}, \text{ tel que } t - B < r_j^i(t) \leq t \quad (2.7)$$

Cette contrainte supprime la possibilité d'avoir des retards arbitrairement grands et implique des temps de communication finis. En pratique, les retards sont toujours bornés et des retards arbitrairement grands correspondent à des pannes (du processeur émetteur ou du lien de communication) qui ne font pas partie des hypothèses de départ. De plus, avec de tels retards, le processus itératif peut être arbitrairement long, ce qui ne présente pas un grand intérêt

pratique. Cette contrainte ne réduit donc pas le champ d'application pratique de l'algorithme et implique que l'on peut évaluer les temps maximaux de communication sur les liens entre processeurs et donc calculer la valeur de `TpsMaxAnnul`.

Enfin, comme l'ordre des communications n'est pas forcément respecté (l'ordre des réceptions n'est pas forcément identique à celui des envois correspondants), le numéro d'itération courante est ajouté aux messages d'état pour permettre une gestion cohérente de ces messages lors des réceptions. Le principe est de ne prendre en compte que les messages qui correspondent à un état plus récent de l'expéditeur, comme on peut le voir dans l'Algorithme 7. Le tableau d'entiers `NumIterMax`, défini uniquement sur le processeur maître, contient pour chaque voisin le plus grand numéro d'itération reçu dans les messages d'état. Ce tableau est initialisé à 0. Comme pour les autres fonctions de communication, cette fonction de réception est exécutée dans un processus léger mais uniquement sur le processeur maître. L'Algorithme 8 détaille le mécanisme principal de détection de la convergence globale.

Algorithme 7 Fonction `ReçoitEtat()`

```

NumSource = Numéro du processeur expéditeur du message
NumIter = Itération à laquelle le message a été envoyé par l'expéditeur
EtatProc = État local de l'expéditeur à l'itération NumIter

Réception des données
si NumIter > NumIterMax[NumSource] alors
    TabEtats[NumSource] = EtatProc
    NumIterMax[NumSource] = NumIter
    si tous les éléments de TabEtats sont Vrai alors
        ConvGlobale = Vrai
    fin si
fin si

```

Une fois la convergence globale détectée sur le processeur maître, il est nécessaire d'arrêter le processus de calcul sur l'ensemble des processeurs. Pour cela, le maître envoie de manière asynchrone un message d'arrêt du calcul à tous les autres processeurs.

Pour terminer proprement l'algorithme, chaque processeur qui reçoit un tel message arrête ses itérations de calcul et affiche ou sauvegarde sa partie du résultat. Puis, il attend que toutes ses communications en cours soient terminées. Cela est possible en testant l'état des mutex relatifs aux communications. Ensuite, une barrière générale est utilisée pour synchroniser tous les processeurs et assurer qu'il n'y a plus de communications en cours dans le système. Puis, le maître peut arrêter le système de communication. Le coût le plus important dans cette procédure d'arrêt, décrite dans l'Algorithme 9, est la barrière de synchronisation. Néanmoins, les résultats expérimentaux obtenus semblent indiquer qu'elle ne détériore pas de manière significative les performances globales de l'algorithme.

Un dernier point qu'il est important de souligner est le fait que la qualité des résultats obtenus avec ce schéma asynchrone est équivalente à celle des résultats obtenus par

Algorithme 8 Détection de la convergence globale

... variables de l'Algorithme 4 ...

Résidu = *Résidu local entre deux itérations consécutives*

Seuil = *Seuil du résidu pour la convergence*

NbSousSeuil = *Nombre d'itérations consécutives où le résidu est au-dessous du seuil, initialisé à 0*

ConvLocaleSup = *Booléen indiquant une convergence locale supposée, initialisé à Faux*

ConvGlobale = *Booléen indiquant la convergence globale, initialisé à Faux*

TabEtats = *Tableau de booléens contenant les états des processeurs, présent uniquement sur le processeur maître et initialisé à Faux*

NumMaître = *Numéro du processeur maître*

NumProc = *Numéro du processeur*

... Initialisations ...

répéter

... Calculs et communications du processus itératif ...

Résidu = $\max_i |NouvellesValeurs[i] - AnciennesValeurs[i]|$

si Résidu > Seuil **alors**

NbSousSeuil = 0

si ConvLocaleSup = Vrai **alors**

si NumProc \neq NumMaître **alors**

ConvLocaleSup = Faux

Envoi asynchrone de ConvLocaleSup au processeur maître

sinon

TabEtats[ProcMaître] = Faux

fin si

fin si

sinon

NbSousSeuil = NbSousSeuil + 1

si NbSousSeuil = NbItCvLocSup **et** NumProc \neq NumMaître **alors**

ConvLocaleSup = Vrai

Envoi asynchrone de ConvLocaleSup au processeur maître

fin si

si NbSousSeuil \geq NbItCvLocSup **et** NumProc = NumMaître **alors**

TabEtats[ProcMaître] = Vrai

si tous les éléments de TabEtats sont Vrai **alors**

ConvGlobale = Vrai

fin si

fin si

fin si

jusqu'à intervalle de temps où ConvGlobale = Vrai > TpsMaxAnnul

si NumProc = NumMaître **alors**

Envoi d'un message de convergence globale à tous les autres processeurs

fin si

... Post-traitements ...

Algorithme 9 Procédure d'arrêt

```

... variables de l'Algorithme 8 ...

... Initialisations ...
répéter
    ... Processus itératif de l'Algorithme 8 ...
jusqu'à intervalle de temps où ConvGlobale = Vrai > TpsMaxAnnul
Affichage ou sauvegarde des données locales
Attente de la fin des communications en cours
Barrière de synchronisation
si NumProc = NumMaître alors
    Arrêt du système de communication
fin si

```

l'homologue synchrone. En effet, puisque le même critère de convergence est utilisé dans les deux cas, les itérations s'arrêtent pour des contextes de résidus similaires. Cela a été confirmé dans toutes nos expérimentations.

Maintenant que le schéma général de calcul est précisé, nous allons voir pour chacun des deux exemples de problèmes scientifiques types, quelles sont les adaptations éventuelles à effectuer et quels sont les résultats expérimentaux obtenus dans des contextes de grappes et/ou de méta-grappes de calcul.

2.3.2 Problème linéaire creux stationnaire

Ce problème classique est de la forme :

$$A.x = b \quad (2.8)$$

où A est une matrice carrée creuse de taille $n \times n$, b est un vecteur connu de taille n et x est le vecteur inconnu de taille n que l'on cherche à calculer.

La méthode choisie pour résoudre un tel problème de manière itérative est la descente de gradient à pas fixe. Son principe est de calculer les approximations successives de x en utilisant l'inverse de la matrice diagonale par bloc M extraite de A . L'idée est que chaque processeur calcule une partie du vecteur x . Pour cela, la matrice A est décomposée en autant de bandes horizontales qu'il y a de processeurs et chaque processeur se voit assigné une bande de la matrice A ainsi que les morceaux des vecteurs b et x qui y correspondent, comme on peut le voir sur la Figure 2.8. En plus de ces données, chaque processeur reçoit aussi la sous-matrice carrée de M contenant le morceau de diagonale correspondant de A .

La formulation globale du processus itératif est alors :

$$x^{k+1} = x^k + \gamma M^{-1}(b - Ax^k) \quad (2.9)$$

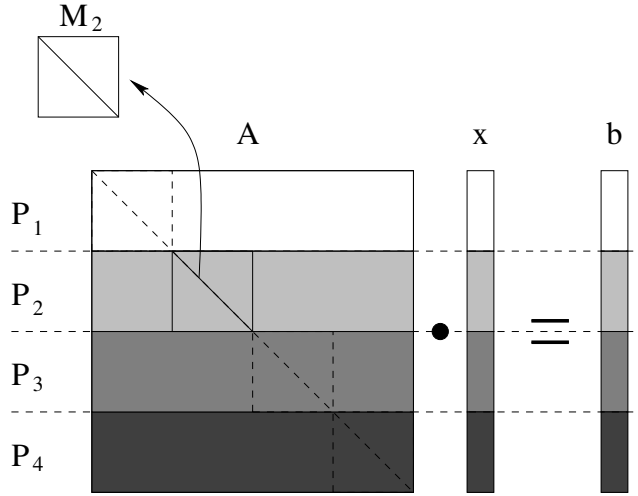


FIG. 2.8 – Décomposition du problème linéaire pour quatre processeurs.

où γ est une constante réelle qui doit être convenablement choisie (autour de 1) pour accélérer la convergence. Lorsque $\gamma = 1$, nous obtenons la méthode de Jacobi. Le processus peut être initialisé avec une valeur arbitraire de x .

Dans le contexte de la décomposition par blocs décrit ci-dessus, le processus 2.9 se réécrit :

$$x_i^{k+1} = x_i^k + \gamma M_i^{-1} (b_i - A_i x^k), \quad i \in \{1, \dots, N\} \quad (2.10)$$

où x_i, M_i, b_i et A_i sont les morceaux respectifs de x, M, b et A associés au processeur i . Les données mises en jeu dans le processus de calcul local à un processeur sont décrites dans la Figure 2.9 dans le cas du processeur 2.

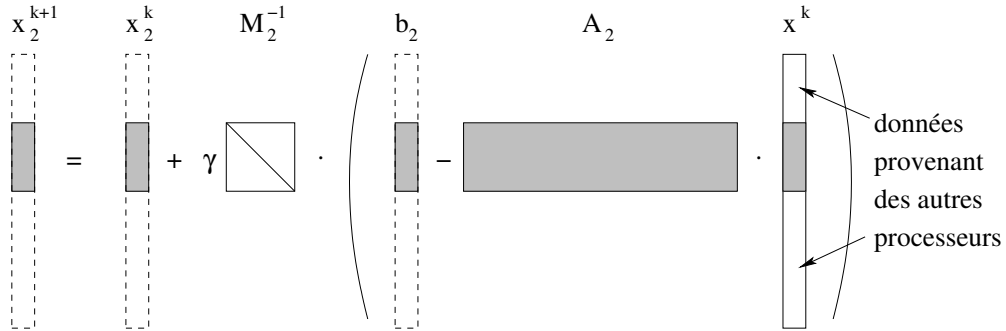


FIG. 2.9 – Processus itératif de résolution du problème linéaire sur le processeur 2.

Ainsi, chaque processeur calcule les approximations successives de son morceau de vecteur x en utilisant les dernières valeurs des données locales ainsi que les dernières valeurs reçues des morceaux de x nécessaires au calcul et provenant des autres processeurs. Bien que la matrice A soit creuse, il est très probable que pour chaque processeur il y ait au moins une valeur dans chaque morceau de x qui soit nécessaire au calcul local. Cela signifie que les dépendances entre processeurs sont du type *all-to-all*. Dès que les valeurs des données locales sont mises à

jour, elles sont envoyées de manière asynchrone aux processeurs qui en ont besoin. Ainsi, à la différence du schéma présenté dans la Figure 2.7, le schéma de communication utilisé ici se rapproche du cas flexible dans lequel il peut y avoir plusieurs envois de données par itération. Par contre, le schéma d'exclusion mutuelle des envois de données vers une même destination est conservé. Enfin, les réceptions des données provenant de processeurs différents sont gérées dans des processus légers séparés.

La solution de (2.8) est donnée par la convergence du processus séquentiel (2.9) ou du processus parallèle (2.10) qui est assurée en asynchrone, comme indiqué précédemment, lorsque $\rho(|I - \gamma M^{-1}.A|) < 1$, où I est la matrice identité. Mis à part un nombre limite d'itérations intégré pour éviter les problèmes de seuil de convergence trop petit, les procédures de détection de convergence et d'arrêt du système sont identiques au schéma général donné précédemment.

La mise en œuvre de l'algorithme asynchrone a été effectuée en utilisant l'environnement de programmation PM2 [118]. Bien que plusieurs environnements puissent être utilisés pour programmer les algorithmes asynchrones, comme nous le verrons en 2.6, celui-ci permet de gérer facilement les processus légers et permet aussi d'éviter autant que possible les copies de données lors des communications. La version synchrone a, quant à elle, été mise en œuvre avec MPI [94].

Dans le contexte de nos expérimentations, trois tailles de matrices ont été testées : 1000000^2 , 2000000^2 et 3000000^2 . Pour chacune d'elles, les valeurs non nulles sont localisées sur trente sous-diagonales. Le seuil de résidu utilisé pour la convergence est fixé à $1e-6$. Le système de calcul est constitué d'une grappe de dix machines réparties sur trois sites distincts. Les liens entre les sites sont de l'Ethernet 20Mb/s alors que les liens intra-site sont de l'Ethernet 100Mb/s. Le premier site contient un Pentium IV 1,6Ghz, deux Pentium IV 2,4Ghz, un Pentium IV 2,6Ghz et un Athlon 2,1Ghz. Le second site contient un Pentium III 833Mhz et le troisième site contient quatre Athlon 1,7Ghz. Bien que le nombre total de machines soit relativement restreint, cette configuration est assez représentative d'une méta-grappe de calcul. Certaines des machines pouvant être utilisées par plusieurs personnes, les temps donnés dans la Table 2.1 sont des moyennes de séries d'exécutions effectuées en veillant à ce que les machines ne soient pas trop chargées. De plus, ces temps ne concernent que la partie de calcul et ne prennent pas en compte les temps de chargement et de sauvegarde des données.

taille de la matrice	version synchrone (MPI)	version asynchrone (PM2)	ratio
1000000×1000000	300,48	203,91	1,47
2000000×2000000	609,23	398,18	1,53
3000000×3000000	1062,39	622,32	1,71

TAB. 2.1 – Temps de calcul (en secondes) de l'algorithme AIAC de résolution du problème linéaire creux sur une méta-grappe de dix machines réparties sur trois sites.

Bien que ce type d'application représente un cas difficile de calcul sur grappes ou méta-grappes à cause des multiples dépendances entre processeurs, on peut constater que la version asynchrone obtient des performances nettement meilleures que la version synchrone. Cela est

très prometteur car la nature des calculs effectués implique que la progression du processus itératif est réduite sur un processeur qui ne reçoit pas de mises à jour de ses dépendances. Malgré tout, l'asynchronisme reste efficace car les retards des messages de données provenant des voisins d'un processeur et destinés à celui-ci sont rarement simultanés. Il reçoit donc constamment des mises à jour d'au moins une partie de ses dépendances qui font progresser ses calculs locaux. Enfin, l'amélioration du gain avec la taille de la matrice montre que la version asynchrone tolère mieux l'augmentation des volumes de données à échanger entre les processeurs.

Ce premier exemple met en valeur le potentiel de l'asynchronisme dans des contextes de méta-grappes de calcul pour des problèmes où les communications sont prépondérantes, c'est-à-dire dans lesquels le rapport entre communications et calculs est relativement important. En effet, dans cet exemple le calcul de chaque itération est assez rapide alors que la quantité de données à échanger est assez importante et que les dépendances entre processeurs sont nombreuses.

2.3.3 Problème non linéaire non stationnaire

Le problème non linéaire que nous avons choisi de résoudre est le problème médical d'Akzo-Nobel [108]. Il consiste en un système raide d'Équations aux Dérivées Partielles (EDPs ou PDEs en anglais) dont la résolution nécessite l'utilisation de méthodes implicites (voir [74]). Ce problème modélise de manière unidimensionnelle la pénétration d'anticorps marqués radio-activement dans un tissu infecté par une tumeur. Les résultats attendus sont donc les évolutions des concentrations des deux éléments (anticorps et tissu) le long de l'espace discrétisé sur un intervalle de temps donné. Ces évolutions, respectivement notées u et v , sont données par :

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - kuv \quad \text{et} \quad \frac{\partial v}{\partial t} = -kuv$$

où k est un coefficient constant.

En utilisant la méthode des lignes, ce problème peut être reformulé par une Équation Différentielle Ordinaire (EDO ou ODE en anglais) de la forme :

$$\frac{dy}{dt} = f(y, t) \quad \text{avec } y(0) \text{ donné} \quad (2.11)$$

où les éléments $y_{2j-1}, j \in [1, P]$ représentent les concentrations d'anticorps (u) le long de l'espace discrétisé en P points et les $y_{2j}, j \in [1, P]$ sont les concentrations de tissu (v). Nous avons donc $y \in \mathbb{R}^{2P}$, $y(0) = (0, v_1(0), \dots, 0, v_P(0)) \in \mathbb{R}^{2P}$ pour conditions initiales et une évolution des y donnée par :

$$\begin{aligned} y_{2j-1} &= \alpha_j \frac{y_{2j+1} - y_{2j-3}}{2\Delta\zeta} + \beta_j \frac{y_{2j-3} - 2y_{2j-1} + y_{2j+1}}{(\Delta\zeta)^2} - k y_{2j-1} y_{2j}, \\ y_{2j} &= -k y_{2j} y_{2j-1}, \end{aligned} \quad (2.12)$$

où

$$\Delta\zeta = \frac{1}{N}, \quad k = 100, \quad \alpha_j = \frac{2(j\Delta\zeta - 1)^3}{c^2}, \quad \beta_j = \frac{(j\Delta\zeta - 1)^4}{c^2} \quad \text{avec } j \in [1..N] \text{ et } c = 4.$$

avec les conditions aux limites :

$$\begin{aligned} y_{-1}(t) &= 2 \\ y_{2N+1}(t) &= y_{2N-1}(t) \end{aligned}$$

Comme on peut le voir, ce problème appartient à la classe des Problèmes à Valeurs Initiales (PVI ou IVP en anglais). C.W.Gear [88] a proposé deux grandes façons de résoudre de tels problèmes en parallèle. La première méthode consiste à résoudre le problème pour chaque pas de temps et s'apparente donc aux algorithmes directs multi-étapes. La seconde méthode correspond aux algorithmes de relaxation d'onde (Waveform Relaxation en anglais) introduits par E.Lelarasemee [107]. Dans ce cas, les éléments à calculer ne sont plus simplement des valeurs simples mais des fonctions définies sur tout l'intervalle de temps. Ces fonctions sont calculées selon les valeurs initiales du problème et les formes courantes des fonctions du système selon les dépendances entre celles-ci.

C'est cette seconde méthode, très bien adaptée au calcul parallèle, qui a été choisie ici. Dans notre cas, les fonctions d'onde sont les éléments y_i , $i \in [1, 2P]$ représentant les évolutions des concentrations u et v sur les points de discrétisation de l'espace. De plus, les résultats de convergence en asynchrone de ces algorithmes [54, 105, 52, 86] nous assurent que l'adaptation AIAC convergera elle aussi. Dans notre cas, le système initial d'EDPs est transformé en un système d'EDOs en utilisant la méthode des différences finies pour discrétiser les dérivées spatiales. Ainsi, nous obtenons une ODE telle que la fonction f est une fonction continue de Lipschitz par rapport à y , c'est-à-dire qu'il existe une constante L telle que :

$$\|f(a, t) - f(b, t)\| \leq L\|a - b\| \quad (2.13)$$

L'inégalité 2.13 assure la convergence uniforme de l'algorithme de relaxation d'onde dans n'importe quel intervalle fini de temps $[t_0, t_1]$.

Pour résoudre (2.11), un algorithme en deux étapes est utilisé :

À chaque itération :

- utilisation de l'algorithme d'Euler implicite pour approcher $\frac{dy}{dt}$
- utilisation de l'algorithme de Newton pour résoudre le système non linéaire obtenu

Les fonctions y_i sont distribuées de manière homogène sur les processeurs en suivant l'organisation linéaire dictée par la dimension du problème. De plus, pour calculer l'évolution des éléments de la discrétisation spatiale sur l'intervalle de temps, ce dernier est aussi discrétisé avec un pas δt fixé par l'utilisateur.

Ainsi, chaque processeur applique la méthode de Newton sur ses éléments locaux en utilisant les données provenant des autres processeurs qui sont nécessaires à ses calculs. Dans le cadre d'un algorithme AIAC, ce sont les dernières valeurs connues de ces données qui

sont utilisées. Le résidu est calculé en utilisant la norme du max sur les fonctions locales y_i entre deux itérations consécutives. Le schéma de communication est assez différent du cas précédent. En effet, d'après la formulation du problème d'Akzo-Nobel, il apparaît que le calcul d'une fonction y_i dépend des deux fonctions avant y_i et des deux fonctions après. Ainsi, si un processeur doit calculer toutes les fonctions de y_p à y_q ($p \leq q$), il aura besoin des deux fonctions avant y_p et des deux fonctions après y_q . Si l'on considère que chaque processeur traite au moins deux fonctions, alors les données extérieures nécessaires à un processeur donné pour effectuer ses calculs proviennent uniquement de ses deux voisins (le précédent et le suivant) dans l'organisation linéaire des processeurs. Ce cas de figure est le plus fréquent en pratique car le nombre de processeurs est généralement bien inférieur au nombre de fonctions y_i à calculer. Dans les cas plus rares où les processeurs ont une seule fonction à calculer, les dépendances s'étendent simplement aux deux processeurs précédents et aux deux processeurs suivants et restent donc dans un voisinage proche. Le schéma de communication s'apparente donc à des échanges locaux entre voisins. Finalement, à chaque itération, chaque processeur envoie de manière asynchrone ses deux premières fonctions à son voisin de gauche et ses deux dernières fonctions à son voisin de droite. Par symétrie, chaque processeur reçoit des données de ses deux voisins. Les réceptions sont gérées par deux fonctions distinctes (une par voisin) qui sont exécutées dans des processus légers séparés. Il est donc nécessaire de spécifier lors de l'envoi de données quelle fonction doit gérer la réception sur le processeur destinataire. En ce qui concerne les mécanismes de détection de convergence et d'arrêt du système, ils ne sont pas modifiés par rapport au schéma général donné précédemment.

En ce qui concerne la mise en œuvre de l'algorithme AIAC, elle a été effectuée avec le même environnement de programmation que celui utilisé pour le cas linéaire, à savoir PM2 [118]. De même, l'environnement MPI a été utilisé pour la version synchrone.

L'environnement de calcul utilisé pour les expérimentations est constitué de douze machines hétérogènes réparties sur quatre sites en France. Les liens de connexion entre les sites est de l'Ethernet 10Mb/s alors que les liens intra-site sont de l'Ethernet 100Mb/s. Le premier site contient deux Pentium II 350Mhz et un Pentium II 450Mhz, le second site quatre Pentium III 450Mhz, le troisième site un Pentium III 833Mhz et le dernier site quatre Pentium III 733Mhz.

Nous avons vu précédemment que le schéma de communication utilisé dans cet algorithme est localisé aux voisins dans l'organisation logique des processeurs. L'ordre des machines dans cette organisation logique a donc une influence importante sur les performances globales. C'est pourquoi deux séries de tests ont été effectuées avec deux organisations logiques différentes. La première organisation est basée sur un ordonnancement des machines par site, c'est-à-dire que l'on a d'abord toutes les machines du premier site, puis toutes les machines du second et ainsi de suite. Dans cette configuration, il y a trois liens de communication qui sont en 10Mb/s alors que les autres sont en 100Mb/s. La seconde organisation correspond à une alternance des sites dans laquelle deux machines logiquement voisines sont physiquement sur deux sites distincts. Cette configuration est a priori moins intéressante que la précédente puisque tous les liens de communication sont en 10Mb/s.

Pour chaque configuration, les algorithmes ont été testés avec trois discrétisations spatiales différentes (valeur de P) : 8000, 18000 et 30000. Les autres paramètres sont fixes et consistent en un intervalle de temps de 4 secondes, un δt de 0,05 et un seuil de résidu de $5e-3$. Les résultats sont donnés dans la Table 2.2.

P=8000			
organisation	version synchrone (MPI)	version asynchrone (PM2)	ratio
par site	461,66	124,87	3,69
alternée	656,55	152,45	4,30
P=18000			
organisation	version synchrone	version asynchrone	ratio
par site	845,48	257,01	3,29
alternée	870,71	255,03	3,41
P=30000			
organisation	version synchrone	version asynchrone	ratio
par site	981,49	418,32	2,34
alternée	1258,19	414,12	3,03

TAB. 2.2 – Temps de calcul (en secondes) de l’algorithme AIAC de résolution du problème d’Akzo-Nobel avec deux organisations logiques d’une même méta-grappe de douze machines réparties sur quatre sites.

La première chose qui saute aux yeux lorsque l’on regarde la Table 2.2 est que non seulement la version asynchrone obtient dans tous les cas des performances meilleures que la version synchrone, mais que ce gain est bien plus important encore qu’avec le problème linéaire. Cela vient en partie du fait que même si un processeur ne reçoit pas de mises à jour de ses dépendances entre deux itérations, la méthode de Newton progresse malgré tout sur celui-ci. Ainsi, ce processus itératif est moins sensible aux retards des communications que celui utilisé pour résoudre le problème linéaire. Nous pouvons donc constater ici toute l’efficacité de l’asynchronisme qui permet d’obtenir un recouvrement implicite des communications par des calculs. Ces bénéfices peuvent être observés dès qu’il y a une hétérogénéité dans le système, qu’elle provienne des liens de communication ou des puissances relatives des machines, mais aussi lors du passage à l’échelle sur une grappe homogène locale. En effet, des tests annexes ont aussi été effectués sur une grappe locale de machines homogènes et les résultats montrent une diminution plus lente de l’efficacité de la version asynchrone par rapport à la version synchrone lorsque le nombre de processeurs augmente.

Le second point intéressant porte sur l’influence de l’organisation des processeurs sur les performances. Il apparaît clairement que la version synchrone est bien plus sensible aux différences de configuration que la version asynchrone. Cela vient du fait que les communications dans la version synchrone ne sont pas recouvertes par des calculs, ainsi la qualité de ces communications influence directement les performances globales. Et comme la qualité des liens est moins bonne dans la seconde configuration du système, les performances globales

sont elles aussi moins bonnes. Par contre, le recouvrement effectué dans la version asynchrone permet d'effacer ces différences dès que la quantité de calculs est suffisamment importante par rapport aux communications.

Enfin, la diminution des ratios entre la version synchrone et la version asynchrone lorsque la taille du problème augmente vient du fait que la quantité de calculs prend une part de plus en plus prépondérante par rapport aux communications. Ainsi, le gain apporté par le recouvrement des communications par des calculs tend à diminuer lorsque la taille du problème augmente. Cependant, cette diminution du ratio reste bien plus faible que l'augmentation de la taille du problème. Ce second exemple confirme bien le potentiel de l'asynchronisme quel que soit le rapport entre la quantité de calculs et les communications.

Ces expérimentations et les précédentes montrent que l'asynchronisme apporte qualité et robustesse des performances aux algorithmes itératifs parallèles. Dans les environnements parallèles homogènes locaux tels que les machines parallèles ou les grappes locales, l'asynchronisme améliore les performances dès lors que les communications deviennent non négligeables par rapport à la quantité de calculs. Dans les contextes de méta-grappes de calcul, l'asynchronisme n'a pas simplement un impact positif sur les performances mais se révèle essentiel. En réalisant un recouvrement implicite des communications par des calculs, il réduit considérablement le surcoût induit par les communications sur des liens hétérogènes dont les performances sont sujettes à de grandes fluctuations.

2.3.4 Discussion sur la notion d'accélération dans les grappes hétérogènes

Dans les résultats expérimentaux précédents, que ce soit pour le problème linéaire ou pour le problème non linéaire, aucune accélération ou efficacité n'ont été données. Cela s'explique par le fait qu'il n'est pas directement représentatif de calculer une accélération sur un ensemble de machines hétérogènes, ayant donc des puissances différentes.

Cependant, si l'on prend par exemple le problème non linéaire et que l'on exécute la version séquentielle de l'algorithme sur la machine la plus puissante de la méta-grappe utilisée, on obtient une accélération quasiment linéaire. Ce résultat n'est pas aussi naturel qu'il y paraît. En effet, comme le système est hétérogène, les machines ont des puissances différentes et l'on peut s'attendre à ce que le système ait une puissance équivalente à la moyenne des puissances des machines. Or, si l'on exécute l'algorithme séquentiel sur une machine ayant cette puissance moyenne, l'accélération obtenue devient alors super-linéaire.

On peut penser dans un premier temps à un problème de cache mémoire. Cependant, même si les problèmes de cache influencent les performances pour des problèmes de très grandes tailles, ce comportement est principalement dû à la structure interne du problème.

Si l'on considère le problème d'Akzo-Nobel, il y a une dépendance spatiale sur l'évolution des éléments y_i dans le processus itératif. Au début, les anticorps sont localisés sur les tous premiers éléments de la discrétisation et l'évolution du système vient de ces éléments. Cela

signifie que la mise à jour d'un élément devient significative (au dessus du seuil de convergence) seulement lorsque ses deux éléments précédents ont eux-mêmes évolués, c'est-à-dire été atteints par l'anticorps. Ainsi, il y a ce que l'on peut appeler une phase de propagation dans le processus itératif avant d'avoir des mises à jour significatives sur tous les éléments.

Dans la version parallèle, les éléments étant distribués sur les processeurs, la phase de propagation s'observe aussi au niveau des processeurs puisqu'un processeur évolue si au moins un des éléments y_i qu'il contient évolue. Cela implique que l'organisation logique des processeurs n'a pas seulement un impact sur les communications, mais aussi sur la vitesse de calcul. En effet, si l'on place en premier des machines globalement plus puissantes que la moyenne du système, alors la phase de propagation sera effectuée plus rapidement que sur un système homogène ayant cette puissance moyenne. On obtient donc une accélération super-linéaire.

Ce phénomène n'est pas anodin car il peut se produire dans de nombreux problèmes non linéaires où il peut y avoir des phases de propagation, notamment dans les problèmes à valeurs initiales. De plus, selon le problème traité, une telle phase de propagation peut se retrouver de manière plus générale dans un ordre quelconque sur l'ensemble des éléments spatiaux. Cela se produit dès qu'un des éléments mis à jour n'évolue pas de manière significative avant un certain contexte global. Il est donc intéressant d'étudier ce phénomène de manière plus précise.

Pour cette étude, deux séries de tests ont été effectuées sur une grappe locale de machines hétérogènes : deux Athlon 1,4Ghz, un Pentium III 750Mhz, un Pentium III 450Mhz et un Pentium II 350Mhz. Pour la première série de tests, les machines sont ordonnées de la plus puissante à la moins puissante alors que pour la seconde série elles sont dans l'ordre opposé. L'utilisation d'un réseau local nous permet d'avoir un meilleur contrôle sur ses performances (latence et bande passante) et donc de conserver les mêmes conditions de communication entre les deux séries de tests. Ainsi, seule la différence d'ordonnancement des machines va générer des différences significatives entre les temps d'exécution. Les résultats sont donnés dans la Table 2.3.

machine utilisée	temps séquentiel (s)	
la plus lente	7039	
la machine de puissance moyenne	2858	
la plus rapide	1794	
ordre logique	temps parallèle (s)	accélération
temps théorique sur une grappe homogène ayant la puissance moyenne	571,78	5
plus rapide → plus lent	368,62	7,75
plus lent → plus rapide	1457	1,96

TAB. 2.3 – Performances de l'algorithme AIAC de résolution du problème d'Akzo-Nobel en séquentiel selon la puissance de la machine et en parallèle selon l'ordre des machines.

On constate bien une différence d'accélération selon la façon dont les machines sont arrangées. L'évaluation théorique de l'accélération de l'algorithme AIAC pour le problème d'Akzo-Nobel obtenue sur une grappe locale de N machines hétérogènes par rapport à une exécution séquentielle sur une machine de puissance P_s , en considérant une distribution homogène des éléments, est la suivante :

$$\frac{P_{min}}{P_s} \times ACC(P_{min}, N) \leq Accélération \leq \frac{P_{max}}{P_s} \times ACC(P_{max}, N) \quad (2.14)$$

où

$$P_{min} = \min_{i=1}^N Puiss(i) \quad \text{et} \quad P_{max} = \max_{i=1}^N Puiss(i)$$

et $ACC(p, n)$ représente l'accélération obtenue sur une grappe de n machines ayant une puissance p par rapport à une exécution séquentielle sur une machine de même puissance p . Enfin, $Puiss(i)$ représente la puissance de la machine i . La borne inférieure correspond à un processus dans lequel il n'y a pas de phase de propagation ou alors dans lequel la phase de propagation est effectuée sur les machines les plus lentes. La borne supérieure est atteinte lorsque l'ensemble du processus itératif est une phase de propagation et les machines les plus rapides sont en premier.

Il apparaît donc que les notions classiquement utilisées en parallélisme doivent être redéfinies dans le cadre des grappes hétérogènes. De plus, on constate aussi une différence de comportement selon que l'on utilise des grappes homogènes ou des grappes hétérogènes. Ces différences, qui peuvent être dues à la structure même du problème traité, peuvent être vues comme des cas de déséquilibre du calcul puisque certains processeurs ne participent pas de manière active à la résolution du problème. Pour cette raison et d'autres que nous allons aborder dans la partie suivante, le recours à l'équilibrage de charge reste indispensable pour obtenir les meilleures performances.

2.4 Équilibrage de charge

Contrairement à une idée qui a longtemps été admise dans le domaine, le fait d'utiliser l'asynchronisme ne dispense pas d'équilibrer la charge de calcul sur les processeurs. En effet, en plus des déséquilibres évoqués précédemment, liés à la structure même du problème, la distribution des données à traiter doit prendre en compte l'hétérogénéité entre les machines. Ainsi, si l'on répartit les données à traiter de manière homogène sur un ensemble de processeurs alors que l'une de ces machines est beaucoup plus lente que les autres, cette dernière mettra beaucoup plus de temps à effectuer la même quantité de calculs que les autres, ce qui ralentira globalement le processus de calcul. On voit bien dans cet exemple simple que le problème de la répartition de la charge de calcul est indépendant de la gestion des communications et des itérations. Ces deux aspects forment en fait deux optimisations différentes et complémentaires que l'on peut apporter à un algorithme itératif parallèle :

- lorsque la charge est convenablement répartie, l'asynchronisme permet un recouvrement efficace des communications par des calculs

- même si les algorithmes AIAC se montrent très efficaces, ils ne prennent pas en compte la distribution de la charge de calcul sur les processeurs.

L'équilibrage de charge peut être appliqué de deux façons différentes classiquement utilisées en parallélisme : soit de manière statique, soit de manière dynamique. Ces deux méthodes ont été abordées dans mes travaux sur ce sujet. Une première étude, présentée dans [5], a permis de mettre en évidence l'intérêt de l'équilibrage de charge dans les algorithmes asynchrones en montrant le gain important obtenu avec un simple équilibrage statique. Ensuite, des études plus complètes sur l'équilibrage dynamique dans les algorithmes asynchrones ont été menées et présentées dans [27, 18] et [7].

2.4.1 Équilibrage statique

Une façon simple de montrer l'intérêt de l'équilibrage de charge dans les algorithmes asynchrones est d'effectuer une comparaison entre une version asynchrone classique avec répartition homogène des données sur les processeurs et une version prenant en compte la puissance relative des machines. Dans ce cas, l'algorithme itératif n'est pas modifié mais les données à traiter sont réparties sur les processeurs en fonction de leur puissance. Un processeur rapide aura donc plus de données à traiter qu'un processeur plus lent. Le nombre de données à assigner au processeur i est obtenu par :

$$Données(i) = \left\lceil \frac{Puiss(i)}{P_T} \cdot D \right\rceil \quad (2.15)$$

où $P_T = \sum_{i=1}^N Puiss(i)$ est la puissance totale disponible, $Puiss(i)$ est la puissance du processeur i et D est le nombre total de données à traiter.

Ce calcul peut être effectué soit hors-ligne en se fondant uniquement sur les puissances théoriques des machines, soit en ligne en effectuant une évaluation de la charge des machines avant de répartir les données et de lancer l'algorithme de calcul.

Dans les expérimentations présentées ici, le second type de répartition est utilisé de façon à prendre en compte la charge des machines au départ du calcul. L'algorithme itératif choisi est celui de la résolution du problème d'Akzo-Nobel présenté dans la partie 2.3.3, exécuté sur la même méta-grappe sans le quatrième site, à savoir : deux Pentium II 350Mhz et un Pentium II 450Mhz sur le premier site, quatre Pentium III 450Mhz sur le second et un Pentium III 833Mhz sur le troisième.

Les résultats, donnés dans la Table 2.4, montrent que même avec un équilibrage statique rudimentaire, les performances de l'algorithme asynchrone sont grandement améliorées. On voit donc bien l'intérêt d'utiliser l'équilibrage de charge avec ce type d'algorithmes. De plus, cette amélioration semble être relativement indépendante de la taille du problème.

De tels résultats sont très encourageants et permettent de penser que des performances encore meilleures peuvent être atteintes en utilisant un algorithme d'équilibrage dynamique. Ce que nous allons voir dans la partie suivante.

taille du problème	sans équilibrage	avec équilibrage	gain
10000	296,85	213,73	28,0%
20000	570,86	418,79	26,6%
30000	927,93	617,05	33,5%

TAB. 2.4 – Temps d'exécution (en secondes) de l'algorithme AIAC de résolution du problème d'Akzo-Nobel avec ou sans équilibrage statique sur une méta-grappe hétérogène de huit machines réparties sur trois sites.

2.4.2 Équilibrage dynamique

Le but de l'équilibrage dynamique de charge est de prendre en compte l'évolution de la puissance de calcul disponible sur chaque machine pendant l'exécution de l'algorithme. En effet, dans les contextes de grappes ou de méta-grappes, il n'est pas rare d'avoir des machines exploitées en multi-utilisateurs et/ou en multi-tâches dont les ressources disponibles varient fortement au cours du temps. Le problème de l'équilibrage dynamique a été largement étudié dans différents contextes [141] et les différentes techniques existantes peuvent être classées selon des critères comme centralisée ou non, statique ou dynamique et synchrone ou asynchrone [101].

Dans le contexte des algorithmes AIAC, on se rend vite compte qu'un équilibrage centralisé et direct n'est pas adapté pour obtenir de bonnes performances. En effet, des échanges globaux et synchrones de données ayant lieu à chaque équilibrage réduiraient le gain apporté par l'asynchronisme pendant les calculs. Ainsi, la technique qui semble la plus adaptée est de travailler localement entre processeurs voisins et de manière itérative. Deux processeurs sont considérés comme voisins s'ils ont des données à échanger pour effectuer leurs calculs. De tels algorithmes décentralisés et itératifs ont été proposés initialement par G.Cybenko [78]. Le principe en est d'équilibrer itérativement la charge de chaque processeur avec ses voisins jusqu'à ce que la charge globale soit équilibrée sur tout le système. On peut noter deux grandes variantes de ces algorithmes que sont la diffusion [102, 91, 78] et l'échange par dimension [122, 91, 101, 78] selon que chaque processeur échange de la charge respectivement avec tous ses voisins simultanément ou avec un seul voisin à la fois. Cependant, ces techniques sont synchrones et ont été principalement développées pour des systèmes de calcul homogènes bien que R.Elsasser et al aient abordé le cas hétérogène dans [81].

Bertsekas et Tsitsiklis ont proposé dans [67] un modèle asynchrone pour l'équilibrage itératif décentralisé. Son principe repose sur le fait que chaque processeur possède l'estimation de sa charge et celles de ses voisins. Régulièrement, le processeur recherche s'il a des voisins moins chargés que lui et si c'est la cas, il leur distribue une partie de sa charge. Les auteurs ont concentré leur travail sur la preuve que ce modèle converge vers une distribution équilibrée de la charge.

Ce modèle a servi de base pour concevoir notre propre algorithme, en réalisant toutefois une adaptation importante aux algorithmes AIAC. En effet, dans le contexte de l'équilibrage des algorithmes AIAC, l'originalité vient du fait que l'asynchronisme apparaît aussi bien au niveau de l'application qu'au niveau de la méthode d'équilibrage. Dans ce contexte, les algorithmes AIAC présentent un avantage supplémentaire, ils sont moins sensibles aux légers déséquilibres que les algorithmes synchrones. Il n'est donc pas primordial d'avoir constamment une répartition optimale des calculs sur l'ensemble des processeurs. Le but est donc plutôt ici d'éviter les trop grands écarts de progression des calculs entre les processeurs.

Pour effectuer cette étude, un autre exemple représentatif des problèmes non linéaires a été choisi. Il s'agit du problème de Brusselator [96] qui modélise une réaction chimique dans un espace unidimensionnel aboutissant à une oscillation des concentrations de certains des éléments. Cette réaction décrit la conversion de deux éléments chimiques A et B en deux autres éléments C et D selon les étapes suivantes :



Il se produit une autocatalyse et lorsque les concentrations de A et B sont maintenues constantes, alors les concentrations de X et Y oscillent dans le temps. Quelles que soient les concentrations initiales de X et Y , la réaction converge vers ce que l'on appelle un cycle limite de la réaction. Cela se caractérise par une boucle fermée lorsque l'on représente graphiquement les concentrations de X en fonction de celles de Y .

De manière similaire au problème d'Akzo-Nobel présenté dans la partie 2.3.3, ce problème correspond à un système raide d'EDPs formulé sous la forme d'un PVI. Les résultats attendus sont les évolutions temporelles des concentrations des deux éléments X et Y , respectivement notées u et v , le long de l'espace discrétisé en P points. Ces évolutions sont données par le système différentiel suivant :

$$\begin{aligned} u'_i &= 1 + u_i^2 v_i - 4u_i + \alpha(P+1)^2(u_{i-1} - 2u_i + u_{i+1}) \\ v'_i &= 3u_i - u_i^2 v_i + \alpha(P+1)^2(v_{i-1} - 2v_i + v_{i+1}) \end{aligned} \tag{2.17}$$

avec $\alpha = \frac{1}{50}$ et les conditions aux limites données par :

$$\begin{aligned} u_0(t) &= u_{P+1}(t) = \alpha(P+1)^2 \\ v_0(t) &= v_{P+1}(t) = 3 \end{aligned}$$

Les conditions initiales sont :

$$\begin{aligned} u_i(0) &= 1 + \sin(2\pi x_i) \text{ avec } x_i = \frac{i}{P+1}, i = 1, \dots, P \\ v_i(0) &= 3 \end{aligned}$$

Là aussi, le système est représenté par un vecteur unique y dans lequel les éléments $y_{2j-1}, j \in [1, P]$ représentent les concentrations de X et les éléments $y_{2j}, j \in [1, P]$ sont les

concentrations de Y . Nous avons donc $y \in \mathbb{R}^{2P}$, $y(0) = (u_1(0), v_1(0), \dots, u_P(0), v_P(0)) \in \mathbb{R}^{2P}$ pour conditions initiales et une évolution sur un intervalle de temps de 10 secondes par pas de discrétisation δt .

En ce qui concerne la méthode de résolution utilisée, elle est très similaire à celle utilisée pour le problème d'Akzo-Nobel puisque le problème de Brusselator a la même structure et les mêmes dépendances de calcul entre les éléments de l'espace discrétisé. En effet, le calcul d'un élément y_i dépend aussi des deux éléments précédents et des deux éléments suivants. La résolution en deux étapes utilisant les algorithmes d'Euler implicite et de Newton est donc reprise pour aboutir à l'Algorithme 10 qui présente la version sans équilibrage.

Algorithme 10 Algorithme AIAC sans équilibrage

```

... variables de l'Algorithme 8 ...
N = Nombre de processeurs
NumProc = Numéro du processeur courant dans l'organisation logique linéaire
AncY, NouvY = Tableaux des éléments  $y_i$  locaux
DebC, FinC = Indices des premier et dernier éléments locaux
T = Durée de l'intervalle de temps pour le calcul de l'évolution des éléments
DebT, FinT = Première (0) et dernière ( $T/\delta t$ ) valeur du temps discrétisé

... Initialisations ...

répéter
  pour j=DebC à FinC faire
    pour t=DebT à FinT faire
      NouvY[j,t] = Résoud(AncY[j,t])
    fin pour
    si j=DebC+2 et NumProc > 0 alors
      si il n'y a pas de communication vers le processeur NumProc-1 en cours alors
        Envoi asynchrone des deux premiers éléments locaux au processeur NumProc-1
      fin si
    fin si
  fin pour
  si NumProc < N-1 alors
    si il n'y a pas de communication vers le processeur NumProc+1 en cours alors
      Envoi asynchrone des deux derniers éléments locaux au processeur NumProc+1
    fin si
  fin si
  Copie de NouvY dans AncY
  Détection de la convergence globale
jusqu'à intervalle de temps où ConvGlobale = Vrai > TpsMaxAnnul
... Post-traitements ...

```

Comme précédemment, cet algorithme est présenté en considérant que les communications (envois et réceptions) sont effectuées dans des processus légers complémentaires au processus

de calcul. Les données à traiter sont réparties de manière ordonnée et homogène sur les processeurs dont l'organisation est logiquement linéaire. Un processeur a donc un voisin de gauche et un voisin de droite hormis les deux processeurs aux extrémités qui n'ont qu'un seul voisin. Les tableaux AncY et NouvY contiennent les mêmes éléments avant et après mise à jour. Pour faciliter l'intégration des dépendances dans les calculs, on ajoute à ces tableaux, en plus des éléments locaux, les deux derniers éléments du voisin de gauche au début et les deux premiers éléments du voisin de droite à la fin. L'organisation obtenue est décrite dans la Figure 2.10 où les données (zones grises) sont représentées sur deux lignes distinctes par souci de clarté. Pour la bonne marche de l'algorithme, cette structure devra être constamment conservée, notamment lors des équilibrages.

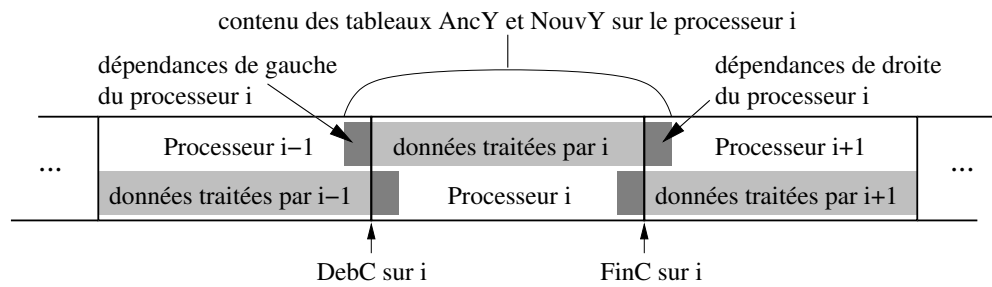


FIG. 2.10 – Organisation des données dans les tableaux AncY et NouvY sur le processeur i .

Pour améliorer les performances de l'algorithme, on peut constater que des communications flexibles exclusives sont utilisées. Ainsi, les données destinées au voisin de gauche lui sont envoyées dès qu'elles ont été mises à jour pendant l'itération. Les données destinées au voisin de droite étant calculées en dernier, leur envoi ne peut se faire qu'à la fin de l'itération comme dans le cas de communications rigides. Lors des réceptions, les données reçues sont directement intégrées dans les calculs en cours. Enfin, pour éviter des incohérences éventuelles lors des communications et des surcharges inutiles du réseau, le système d'exclusion empêchant plusieurs occurrences d'un même type de communication sur un même processeur est mis en place. La Figure 2.11 présente le schéma de communication résultant.

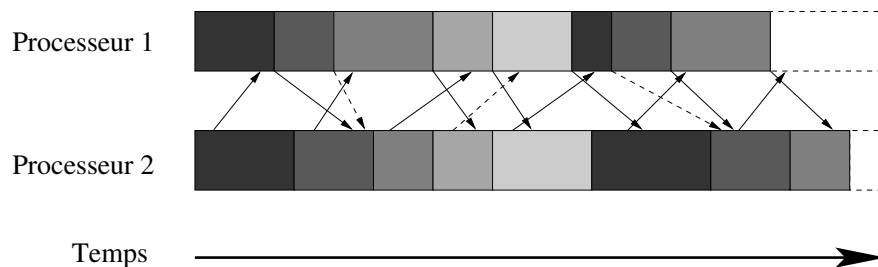


FIG. 2.11 – Schéma d'exécution d'un algorithme AIAC avec communications flexibles exclusives. Les flèches en pointillés représentent les communications non effectuées à cause de l'exclusivité des communications. Les flèches solides commençant en cours d'itération correspondent aux envois des données vers le voisin de gauche alors que les flèches commençant en fin d'itération correspondent aux envois des données vers le voisin de droite.

Une variante évoquée par Bertsekas et Tsitsiklis est de transférer de la charge uniquement vers le voisin le moins chargé. Cette variante a été retenue dans notre algorithme car elle présente l'avantage de ne générer que des transferts de charge par couples de voisins. Cela rend le schéma d'équilibrage plus simple à mettre en œuvre mais aussi et surtout beaucoup plus léger en évitant d'encombrer le réseau avec de trop nombreux messages de charge.

Notre schéma d'équilibrage étant donc local à deux processeurs, la Figure 2.12 décrit son déroulement temporel entre les deux processeurs concernés. Il ne faut pas perdre de vue que ce schéma peut s'appliquer simultanément à plusieurs couples de processeurs dans le système. Par souci de clarté, seuls les envois de données à la fin des itérations sont représentés. Cependant, selon le problème traité, il peut y avoir des envois de données pendant les itérations, comme dans le cas de la Figure 2.11 pour le problème de Brusselator. Le processus d'équilibrage est indépendant du schéma de communication utilisé pour les échanges de données, à ne pas confondre avec les transferts de charge réalisés pour l'équilibrage.

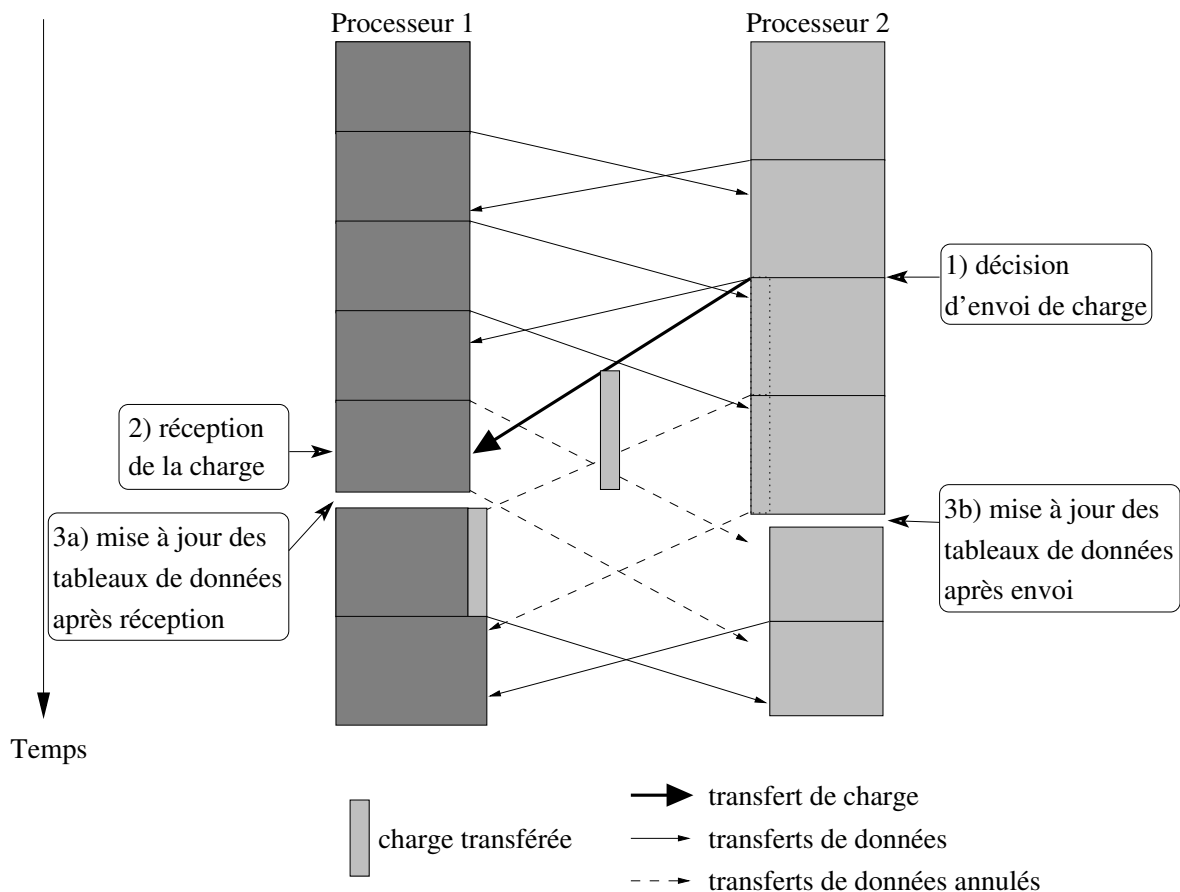


FIG. 2.12 – Schéma d'exécution de l'algorithme d'équilibrage entre deux processeurs.

Dans ce schéma d'équilibrage, chaque processeur vérifie périodiquement s'il doit équilibrer sa charge avec un de ses voisins. La période de test est exprimée en un nombre d'itérations ajustable selon le problème traité. En effet, la fréquence des équilibrages peut être assez

importante dans certains cas alors que des fréquences plus faibles seront recommandées dans d'autres cas pour ne pas augmenter de manière prépondérante le surcoût d'équilibrage, particulièrement lorsque l'on a des liens de communications assez lents.

Lorsqu'un équilibrage est nécessaire, le processeur qui le détecte envoie une partie de sa charge de calcul, c'est-à-dire une partie des éléments qu'il doit traiter, à un voisin moins chargé. Cette étape correspond à la bulle (1) dans la Figure 2.12. Pour prendre en compte ce transfert de charge, chaque processeur doit vérifier à chaque itération si un tel transfert est en cours ou non, ce qui revient à détecter non seulement les envois de charge mais aussi les réceptions de charge, comme indiqué par la bulle (2) sur la figure. Lorsqu'un tel transfert est détecté, les tableaux locaux contenant les données à traiter doivent être redimensionnés de façon à prendre en compte l'ajout ou la suppression de charge. Ces tableaux doivent être agrandis sur le processeur qui reçoit la charge à l'itération suivante la réception de façon à pouvoir y insérer les nouvelles données à traiter, comme indiqué par la bulle (3a). De manière symétrique, ces tableaux doivent être réduits sur le processeur qui envoie la charge de façon à ne pas conserver en mémoire des données qui ne seront plus traitées sur ce processeur. Néanmoins, pour ne pas ralentir l'évolution des éléments transférés, leur suppression n'est effectuée sur l'expéditeur que lorsque leur envoi est complètement terminé, comme indiqué par la bulle (3b). Cette optimisation de la gestion mémoire est un peu délicate et coûteuse mais elle est toutefois nécessaire pour maintenir une consommation minimale de mémoire et assurer qu'il n'y ait pas de saturation sur l'un des processeurs. Cela est notamment important pour les problèmes de grande taille. Enfin, une fois que les tableaux ont été redimensionnés et que les données transférées sont intégrées à leur nouvel emplacement, le processus itératif peut reprendre avec la nouvelle répartition de charge entre les deux processeurs.

Dans le cadre du problème de Brusselator, comme mentionné précédemment, il est important de conserver une répartition globale ordonnée des éléments à traiter sur l'ensemble des processeurs organisés linéairement, comme indiqué dans la Figure 2.10. Et comme dans ce contexte, un processeur a au plus deux voisins, celui de gauche et celui de droite, il y a donc deux cas de figure symétriques pour les transferts. La Figure 2.13 donne le détail des mouvements de données générés dans le cas d'un transfert de charge vers le voisin de gauche.

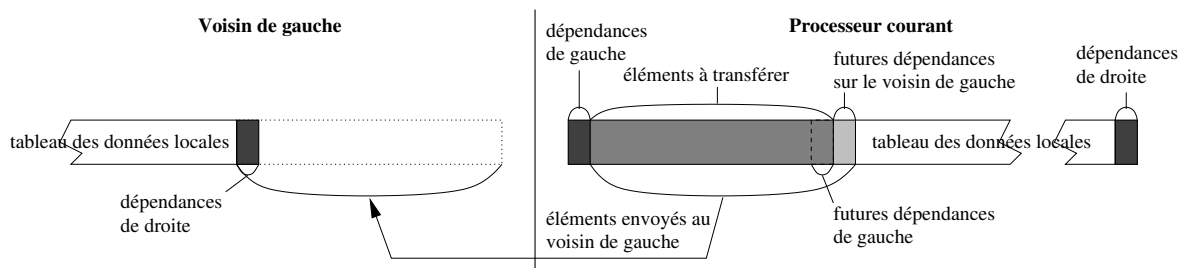


FIG. 2.13 – Transfert de données d'un processeur vers son voisin de gauche.

Comme indiqué sur cette figure, ce sont les premiers éléments locaux, exceptés ceux qui représentent les dépendances de gauche, qui sont transférés au voisin de gauche et ajoutés

à la fin de son tableau local pour conserver l'ordre logique global des éléments. En plus de ces éléments, les valeurs des deux éléments suivants sont aussi envoyées car elles correspondent aux futures dépendances du voisin de gauche après l'équilibrage. Une fois le transfert réalisé, ces deux éléments sont toujours calculés sur le processeur courant et leurs valeurs sont régulièrement envoyées au voisin de gauche pour que celui-ci puisse effectuer ses propres calculs. D'une manière similaire, les nouvelles dépendances du processeur courant sont les deux derniers éléments effectivement transférés. Leurs valeurs sont donc conservées localement et leurs mises à jour régulièrement reçues du voisin de gauche.

L'Algorithme 11 correspond à la version équilibrée de l'Algorithme 10. Seules sont mises en évidence les parties relatives à l'équilibrage de charge. On peut constater que les traitements supplémentaires pour l'équilibrage de charge sont placés au début de la boucle principale du processus itératif. Cela permet de prendre en compte tout transfert de charge avant de commencer une nouvelle itération. On peut aussi noter l'exclusivité des tests d'équilibrage avec les voisins, réalisés dans les fonctions `TestEquGauche()` et `TestEquDroite()`, pour assurer un transfert de charge vers une seule destination à la fois. De plus, ces tests sont effectués de manière périodique et seulement s'il n'y a pas déjà un transfert en cours pour éviter toute incohérence. Le fait que le test d'équilibrage à gauche soit effectué avant celui de droite n'avantage pas de manière significative l'équilibrage à gauche et n'enlève rien à la généralité de notre algorithme. Les fonctions d'équilibrage sont exécutées dans des processus légers pour ne pas bloquer le processus de calcul et obtenir ainsi un algorithme d'équilibrage asynchrone. Enfin, la suite de l'algorithme est similaire à l'Algorithme 10 à ceci près que des informations supplémentaires sont nécessaires lors des communications des dépendances de calcul. En effet, comme les tableaux locaux des données peuvent changer, la position globale des données envoyées est nécessaire pour que le processeur qui reçoit ces données puisse vérifier qu'elles correspondent bien aux dépendances attendues. De même, l'évaluation de la charge du processeur, nécessaire à ses voisins pour pouvoir prendre les décisions d'équilibrage, est ajoutée à ces données.

Les deux fonctions d'équilibrage étant symétriques, seule la fonction `TestEquGauche()` est détaillée dans l'Algorithme 12. La première étape consiste à tester si le déséquilibre entre le processeur et son voisin de gauche est suffisamment important pour effectuer un transfert de charge. Pour cela, le ratio des charges des deux processeurs est calculé et comparé à un seuil fixé. Lorsque le ratio est au-dessus du seuil, alors le nombre d'éléments à envoyer vers le voisin de gauche est calculé en veillant à ce qu'il reste suffisamment de données localement pour éviter les phénomènes de famines. Les données envoyées sont celles précédemment décrites dans la Figure 2.13. Pour être sûr que le transfert des données est bien terminé avant de réduire le tableau sur l'expéditeur, l'envoi est effectué de manière bloquante. Cependant, cela n'enlève en rien la nature asynchrone des transferts de charge par rapport au processus de calcul puisque les fonctions gérant ces transferts sont exécutées dans des processus légers distincts de celui des calculs.

Sur le voisin de gauche, les données transférées proviennent du voisin de droite, c'est donc la fonction `RecEquDroite()` décrite dans l'Algorithme 13 qui effectue leur réception.

Algorithme 11 Algorithme AIAC avec équilibrage

... variables de l'Algorithme 10 ...

EnvCharge = *Booléen indiquant qu'un transfert de charge est en cours depuis le processeur, initialisé à Faux*

EnvChargeFini = *Booléen indiquant que le transfert de charge depuis le processeur est fini, initialisé à Faux*

RécCharge = *Booléen indiquant la réception de charge additionnelle sur le processeur, initialisé à Faux*

NbIterEqu = *Entier indiquant le nombre d'itérations à faire avant de tester si un équilibrage est nécessaire, initialisé à PériodeEqu > 0*

... Initialisations ...

répéter

si RécCharge=Vrai **ou** EnvChargeFini=Vrai **alors**

si RécCharge=Vrai **alors**

 Agrandir NouvY et AncY de façon à pouvoir y ajouter les nouvelles données reçues

 Compléter AncY avec ces nouvelles données

 RécCharge=Faux

fin si

si EnvChargeFini=Vrai **alors**

 Réduire NouvY et AncY de façon à en supprimer les données transférées

 EnvChargeFini=Faux

fin si

sinon

si NbIterEqu=0 **alors**

si EnvCharge=Faux **alors**

 Exécution de TestEquGauche() dans un processus léger

si EnvCharge=Faux **alors**

 Exécution de TestEquDroite() dans un processus léger

fin si

fin si

 NbIterEqu=PériodeEqu

sinon

 NbIterEqu=NbIterEqu-1

fin si

fin si

... Calculs et gestion des envois de données (à gauche et à droite) similaires ...

... à l'Algorithme 10 ...

... Par contre, les données envoyées sont complétées par leur position globale dans ...

... l'ordre logique des données et l'évaluation de la charge à l'itération précédente ...

Détection de la convergence globale

jusqu'à intervalle de temps où ConvGlobale = Vrai > TpsMaxAnnul

... Post-traitements ...

Algorithme 12 Fonction TestEquGauche() /* symétrique à TestEquDroite() */

ChargeLoc = *Évaluation de la charge locale*
 ChargeGauche = *Évaluation de la charge du voisin de gauche*
 Ratio = *Ratio des évaluations de charge entre le processeur et son voisin de gauche*
 NbElts = *Nombre de données traitées actuellement par le processeur*
 NbEnv = *Nombre de données à transférer au voisin de gauche*

Ratio=ChargeLoc/ChargeGauche
si Ratio>SeuilRatio **alors**
 Calcul de NbEnv en fonction de Ratio
 si NbElts-NbEnv>SeuilElts **alors**
 NbIterEqu=PériodeEqu
 Envoi bloquant des NbEnv+2 premiers éléments au processeur NumProc-1
 /* le +2 permet d'inclure les dépendances */
 EnvChargeFini=Vrai
fin si
fin si

Cette fonction consiste simplement à recevoir les données et à les placer dans un tableau temporaire avant leur intégration dans le tableau des données traitées localement. Une fois la réception terminée, l'indicateur booléen est activé pour que l'intégration puisse avoir lieu dans le processus global.

Algorithme 13 Fonction RecEquDroite() /* symétrique à RecEquGauche() */

Réception du nombre de données transférées
 Réception des données et stockage dans un tableau temporaire
 RécCharge=Vrai

Algorithme 14 Fonction RecDonGauche() /* symétrique à RecDonDroite() */

Réception des deux éléments du processeur NumProc-1 et de leur position globale
si position globale des deux éléments = celle des dépendances attendues **alors**
 si aucun redimensionnement en cours sur AncY **alors**
 Copier les deux éléments reçus au début du tableau AncY
fin si
fin si
 Réception de l'évaluation de charge du processeur NumProc-1

Pour terminer, les fonctions de réception des dépendances de calcul doivent être un peu modifiées. En effet, comme mentionné précédemment, il est nécessaire de vérifier que les données reçues correspondent bien aux dépendances locales attendues. Cela est fait en utilisant la position globale qui est jointe aux données. La copie de ces données dans le tableau des données locales n'est effectuée que si ce tableau n'est pas en cours de redimensionnement. Enfin, la réception de l'évaluation de charge jointe aux données doit être ajoutée. L'algo-

rithme 14 décrit la fonction de réception des dépendances de calcul provenant du voisin de gauche.

Finalement, étant donné que le processus d'équilibrage ne modifie pas le processus de calcul, le comportement de l'algorithme AIAC reste similaire entre les deux versions. Ainsi, si l'algorithme asynchrone sans équilibrage converge, alors la convergence de sa version équilibrée sera assurée.

Un dernier point qui reste à régler pour que notre algorithme soit complet est le choix de l'estimateur de charge. En général, la charge d'un processeur est évaluée en calculant le temps nécessaire pour traiter une quantité fixe de données du problème (un élément de la discrétisation spatiale dans notre exemple). La redistribution des données effectuées en utilisant cette estimation de charge a pour effet d'obtenir approximativement les mêmes temps de calcul d'une itération sur tous les processeurs. Cependant, cet équilibrage n'est efficace que si tous les éléments ont approximativement la même progression dans le processus itératif. En effet, le point essentiel pour l'efficacité des algorithmes itératifs parallèles est que tous les processeurs atteignent leur convergence locale à peu près au même moment. Or, comme nous l'avons vu précédemment avec l'exemple du problème d'Akzo-Nobel, il arrive que la progression vers la solution ne soit pas la même pour tous les éléments du système. À certains moments, certains éléments vont progresser de manière importante alors que les autres ne progresseront pas de manière significative et à d'autres moments, ce sont d'autres éléments qui vont progresser. Pour arriver au résultat souhaité, il faut donc prendre en compte la progression relative des éléments du problème. Cela n'est possible qu'en utilisant le résidu entre deux itérations consécutives. Comme on utilise la norme du max, le résidu utilisé pour estimer la charge d'un processeur est le max des résidus de ses éléments locaux.

Ainsi, lorsque deux processeurs voisins ont des résidus très différents, cela signifie que l'un progresse moins que l'autre et a donc une contribution plus faible à la progression globale du calcul. Pour équilibrer les contributions, on effectue alors un transfert d'éléments du processeur ayant le plus grand résidu vers celui ayant le plus petit résidu. Cette action vise à mieux répartir sur les processeurs les éléments ayant des résidus importants et donc progressant de manière importante. Cependant, comme l'ordre global des éléments doit être conservé lors des transferts de charge, il n'est pas possible de choisir précisément les éléments à transférer. Il n'est donc pas assuré que les éléments transférés soient ceux ayant des résidus importants. Malgré tout, ce transfert de charge reste utile car il permet aussi d'accélérer la progression sur le processeur ayant cédé des éléments en allégeant sa charge locale. En effet, ayant moins d'éléments à traiter, ce processeur va effectuer ses itérations plus vite et va donc progresser plus vite. Son résidu va donc décroître plus rapidement et combler finalement son retard de progression sur son voisin.

Ce mécanisme et son impact sur l'évolution des résidus des processeurs est schématisé dans la Figure 2.14. Les rectangles gris sur les côtés représentent les itérations successives sur les processeurs et les courbes donnent les résidus des éléments locaux. On peut constater qu'au début de la séquence, les processeurs ont approximativement le même nombre de données et le

calcul d'une itération prend à peu près le même temps. Cependant, la différence importante entre les résidus des processeurs déclenche un équilibrage. Cela ne serait pas le cas si l'on utilisait l'estimateur classique. Une fois le transfert de charge effectué, le processeur 1 effectue ses itérations plus vite puisqu'il a moins d'éléments alors qu'au contraire, les itérations du processeur 2 sont ralenties par le plus grand nombre d'éléments. Le résidu du processeur 1 a donc tendance à diminuer plus rapidement pour finalement rattraper celui du processeur 2 au bout de quelques itérations.

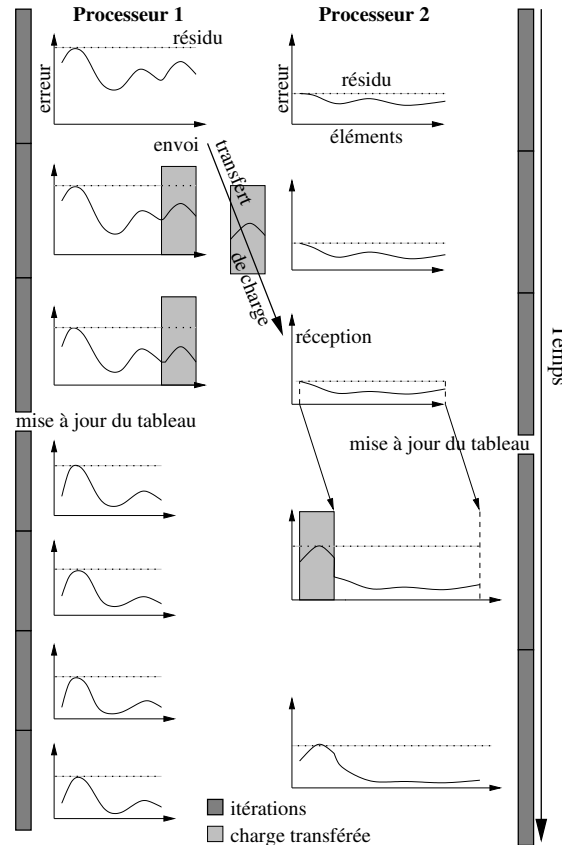


FIG. 2.14 – Transfert de charge entre deux processeurs, basé sur le résidu.

Pour mettre en évidence l'impact du choix de l'estimateur de charge sur les performances, les temps d'exécution de l'algorithme avec équilibrage sont comparés en fonction de la précision demandée selon que l'on utilise l'estimateur classique ou le résidu. La mise en œuvre a été effectuée avec le même environnement de développement que dans les parties précédentes. Les paramètres utilisés sont 60000 points de discrétisation, un δt de 0,05 secondes et un seuil de résidu de $7e-4$. Enfin, le contexte de calcul utilisé est une méta-grappe de dix machines, allant d'un Pentium II 450Mhz à un Pentium IV 2,4Ghz, réparties sur trois sites possédant des réseaux locaux à 100Mb/s reliés entre eux par des liens à 10Mb/s.

Les résultats présentés dans la Figure 2.15 sont des moyennes de vingt exécutions. Ils montrent clairement que l'utilisation du résidu pour estimer la charge apporte un gain de

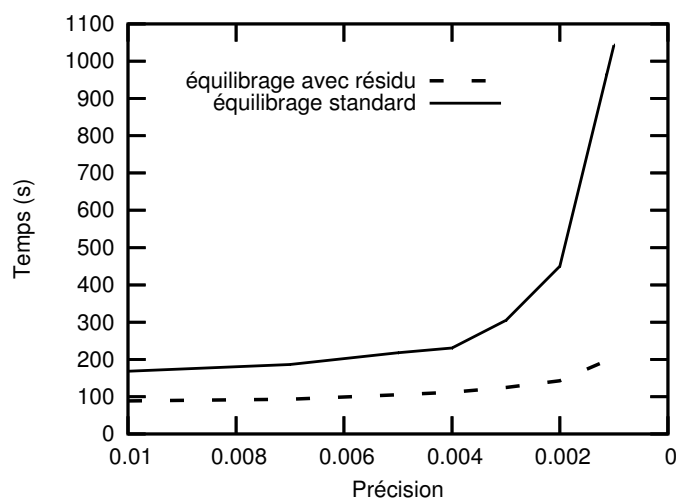


FIG. 2.15 – Temps d'exécution de l'algorithme AIAC de résolution du problème de Brusselator 1D équilibré avec deux estimateurs de charge différents en fonction de la précision demandée.

performances important, particulièrement avec les grandes précisions pour lesquelles un nombre plus important d'itérations est nécessaire. Cela s'explique par le fait qu'avec l'estimateur classique, la probabilité d'avoir des processeurs qui participent peu à la progression du calcul global augmente régulièrement tout au long du processus itératif. Ainsi, le déséquilibre entre les progressions des processeurs augmente considérablement avec le nombre d'itérations dans le processus.

Maintenant qu'il a été établi que le résidu est l'estimateur de charge le plus adapté à l'équilibrage des algorithmes AIAC, une comparaison des deux versions de l'algorithme, avec ou sans équilibrage, s'impose. Cette comparaison est menée dans deux contextes de calcul, une grappe homogène locale composée de Pentium III 733Mhz reliés par une réseau à 100Mb/s et une méta-grappe de quinze machines allant d'un Pentium II 400Mhz à un Athlon 1,6Ghz, réparties sur trois sites reliés entre eux par des liens à 10Mb/s. Là aussi, les résultats obtenus sont des moyennes de vingt exécutions.

La Figure 2.16 donne les comportements des deux versions de l'algorithme, avec ou sans équilibrage, sur la grappe locale en fonction du nombre de processeurs. On peut constater un comportement très similaire des deux versions, à des niveaux de performance différents. Cela est déjà très intéressant car il n'est pas rare que les algorithmes d'équilibrage induisent des surcoûts qui limitent le passage à l'échelle, ce qui n'est pas le cas ici. Mais le plus important est le décalage vertical entre les deux courbes qui dénote un gain important de performances en faveur de la version équilibrée. Le ratio entre les temps d'exécution varie de 6,2 à 7,4 avec une moyenne de 6,8. Ces très bons résultats viennent d'un part, de la nature décentralisée et asynchrone de l'algorithme d'équilibrage et d'autre part, de l'utilisation du résidu comme estimateur de charge.

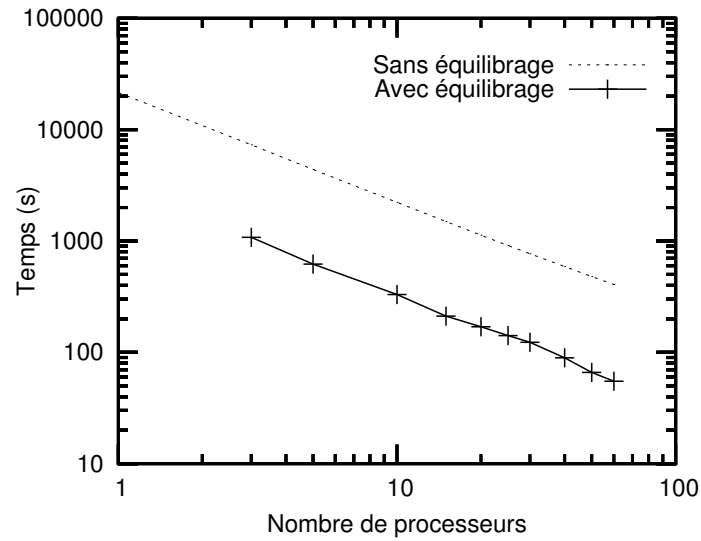


FIG. 2.16 – Temps d’exécution de l’algorithme AIAC de résolution du problème de Brusselator 1D avec ou sans équilibrage sur une grappe homogène locale en fonction du nombre de processeurs.

En ce qui concerne la comparaison sur la méta-grappe, les machines ne sont pas organisées de manière ordonnée, site par site, mais de manière désordonnée entre les sites de façon à obtenir un contexte moins favorable à l’équilibrage de charge. De plus, l’organisation est faite de telle sorte que deux machines voisines n’ont pas des puissances similaires. Les résultats obtenus sont donnés dans la Table 2.5.

version	sans équilibrage	avec équilibrage	ratio
temps d’exécution (s)	515,3	105,5	4,88

TAB. 2.5 – Temps d’exécution (en secondes) de l’algorithme AIAC de résolution du problème de Brusselator 1D avec ou sans équilibrage sur une méta-grappe hétérogène.

Là aussi, on peut constater un gain considérable par rapport aux performances de l’algorithme AIAC initial. Ce gain est plus modéré que dans le cas de la grappe homogène locale du fait des coûts plus élevés des communications et donc des transferts de données. Néanmoins, c’est bien dans le contexte des méta-grappes que le couplage de l’asynchronisme et de l’équilibrage de charge se révèle le plus intéressant. En effet, les algorithmes asynchrones sans équilibrage étant déjà nettement supérieurs à leurs homologues synchrones dans ce contexte, et le gain apporté par l’équilibrage de charge étant malgré tout très important, il en résulte que les algorithmes asynchrones équilibrés sont les plus performants des algorithmes itératifs parallèles dans le contexte des méta-grappes.

Cependant, l’efficacité de l’équilibrage des algorithmes AIAC peut varier selon le problème traité et l’état du réseau utilisé. Les points essentiels pour assurer cette efficacité ont été mis

en évidence. Le premier porte sur le nombre d'itérations qui doit être suffisamment important pour qu'il vaille la peine d'équilibrer la charge. De même, le temps moyen de calcul d'une itération doit être suffisamment important pour avoir un ratio raisonnable de calculs par rapport aux communications. Dans le cas contraire, l'équilibrage améliorerait assez peu les performances tout en surchargeant le réseau de communication. Un autre point important est la fréquence des équilibrages qui doit être ni trop élevée ni trop faible. Cela implique de disposer d'une mesure précise des déséquilibres éventuels entre processeurs mais aussi de bien choisir la précision de l'équilibrage. Lorsque le réseau est chargé ou lent, il est préférable d'effectuer un équilibrage grossier qui sera moins réactif aux légers déséquilibres et provoquera moins de transferts de données. Par contre, un équilibrage plus précis aura tendance à accélérer la progression du calcul. La difficulté est donc de trouver le bon compromis entre ces deux contraintes. Une possibilité est de régler la fréquence et la précision des équilibrages en fonction de la vitesse du réseau.

Finalement, on voit bien que l'asynchronisme et l'équilibrage de charge ne sont pas incompatibles et n'apportent pas les mêmes optimisations. Leur couplage est donc essentiel pour obtenir les algorithmes les plus efficaces. De plus, nous avons vu que l'utilisation du résidu comme estimateur de charge permet de veiller à ce que tous les processeurs contribuent à la progression du processus itératif.

2.5 Détection décentralisée de la convergence globale

Dans les parties précédentes, la détection de la convergence globale est réalisée par l'Algorithme 8 qui est centralisé. Un processeur reçoit les informations d'état des autres processeurs et détecte la convergence globale lorsque tous les états sont en convergence locale. Cependant, même si cette version centralisée utilise des communications asynchrones, elle n'est pas la plus adaptée aux algorithmes AIAC. En effet, le processus de calcul et l'algorithme d'équilibrage étant décentralisés, il paraît judicieux d'utiliser une détection de convergence qui soit elle aussi décentralisée. De plus, dans les contextes de méta-grappes, il n'est pas rare d'être confronté à des restrictions d'accès entre les différents sites formant la méta-grappe. Ainsi, il arrive que les processeurs ne puissent pas communiquer avec tous les autres, rendant un algorithme centralisé inutilisable. Il existe bien des techniques de transmission indirecte (forwarding en anglais) permettant de conserver un schéma centralisé, mais leurs performances sont dégradées par la surcharge des liens de communication entre les sites. Décentraliser la détection de convergence permet donc de grandement faciliter et améliorer le déploiement des algorithmes AIAC quelle que soit la méta-grappe utilisée. L'objectif est de n'avoir que des communications locales entre processeurs voisins et un délai de détection de la convergence le plus petit possible.

Ce problème est assez similaire à celui de la terminaison distribuée qui consiste à trouver le moment auquel un calcul distribué se termine. Ce problème a bien entendu été étudié dans le contexte du calcul distribué [83, 79, 121]. Néanmoins, la plupart des travaux menés sont fondés

sur des algorithmes centralisés ou dans des contextes synchrones [111]. En ce qui concerne les algorithmes itératifs asynchrones, la détection de terminaison distribuée a été présentée initialement dans [66] sous certaines conditions relativement restrictives. S.A.Savari et Bertsekas ont proposé une autre version distribuée dans [128] mais avec des hypothèses encore plus restrictives telles que des communications ordonnées et des modifications du processus itératif pour en assurer la terminaison. D'autres auteurs ont étudié la mise en œuvre d'algorithmes asynchrones mais toujours en utilisant une détection de convergence centralisée [75]. Une revue des différentes méthodes utilisées pour la terminaison des algorithmes itératifs asynchrones sur des systèmes à passage de message est disponible dans [61].

Le problème majeur pour détecter la convergence globale de manière décentralisée dans les algorithmes asynchrones vient justement de l'asynchronisme qui induit une progression vers la convergence bien plus complexe que dans le cas synchrone. Cela peut impliquer une détection tardive de la convergence qui augmente inutilement le temps d'exécution de l'algorithme, ou pire, une détection prématurée et donc erronée.

Mon travail sur ce problème a abouti à un nouvel algorithme décentralisé de détection de la convergence globale présenté dans [8]. L'originalité de l'approche utilisée dans cette étude est d'utiliser l'algorithme de l'élection de leader [110] (leader election en anglais) pour obtenir un algorithme fiable de détection décentralisée. Cet algorithme d'élection consiste à choisir dynamiquement un processeur pour effectuer une tâche particulière et est généralement utilisé dans les protocoles de communication tels que le firewire IEEE 1394 [131] ou dans les systèmes distribués [47, 80]. Il a aussi été utilisé dans des systèmes asynchrones [87, 132] mais dans des contextes de tolérance aux pannes et non pour détecter la terminaison d'une application.

La partie 2.5.1 décrit l'algorithme théorique de détection de convergence fondé sur l'élection de leader. Une version pratique adaptée aux algorithmes AIAC en est déduite et comparée expérimentalement à deux versions centralisées dans la partie 2.5.2.

2.5.1 Version théorique

L'algorithme proposé ici peut fonctionner avec le type le plus général des algorithmes itératifs parallèles, à savoir les algorithmes totalement asynchrones. Il peut donc aussi être utilisé avec les autres types d'algorithmes itératifs en effectuant quelques adaptations mineures.

Nous avons vu dans la partie 2.3.1 que le principal problème, commun à tous les algorithmes itératifs parallèles, pour détecter la convergence globale réside dans le fait de détecter convenablement la convergence locale. Cependant, dans ce contexte théorique, la possibilité d'utiliser la constante $NbItCvLoc$ simplifie grandement l'algorithme. En effet, lorsqu'un processeur a effectué au moins $NbItCvLoc$ itérations consécutives sous le seuil de convergence, il est assuré d'avoir atteint la convergence locale et son état local ne changera plus. Il n'y a donc pas besoin des messages de changement d'état de l'Algorithme 8. C'est pour cette raison que cet algorithme théorique n'est pas restreint à des retards bornés.

Dans ce contexte, la difficulté se situe plus dans l'algorithme d'élection de leader utilisé pour la détection de la convergence globale. Son bon fonctionnement nécessite un arbre comme graphe de communication. Lorsque les dépendances du problème traité impliquent un autre type de graphe, il est alors nécessaire d'extraire un arbre couvrant qui sera utilisé uniquement pour le processus de détection de convergence. Le graphe initial est conservé pour les communications liées au processus itératif et à l'équilibrage éventuel.

Le processus d'élection est basé sur l'envoi de messages dits de convergence partielle. Un tel message indique au processeur qui le reçoit que tous les processeurs dans le sous-arbre derrière l'expéditeur ont atteint leur convergence locale. Ainsi, un processeur envoie un tel message lorsqu'il a atteint sa convergence locale et qu'il ne lui reste plus qu'un seul voisin dont il n'a pas reçu un message de ce même type. C'est alors à ce dernier voisin que le message est envoyé. Le fait d'utiliser un arbre couvrant assure qu'il y aura au moins un processeur avec un seul voisin d'où pourra commencer le processus d'élection. Ces messages vont se propager à mesure des convergences locales des processeurs et celui qui aura reçu de tels messages de tous ses voisins lors de sa propre convergence locale sera l'élu. La convergence globale est alors détectée sur ce processeur qui va en informer ses voisins qui eux-mêmes propageront la nouvelle à leurs autres voisins et ainsi de suite sur tout le graphe de communication. La Figure 2.17 schématise le déroulement de cet algorithme et son pseudo-code est fourni dans l'Algorithme 15.

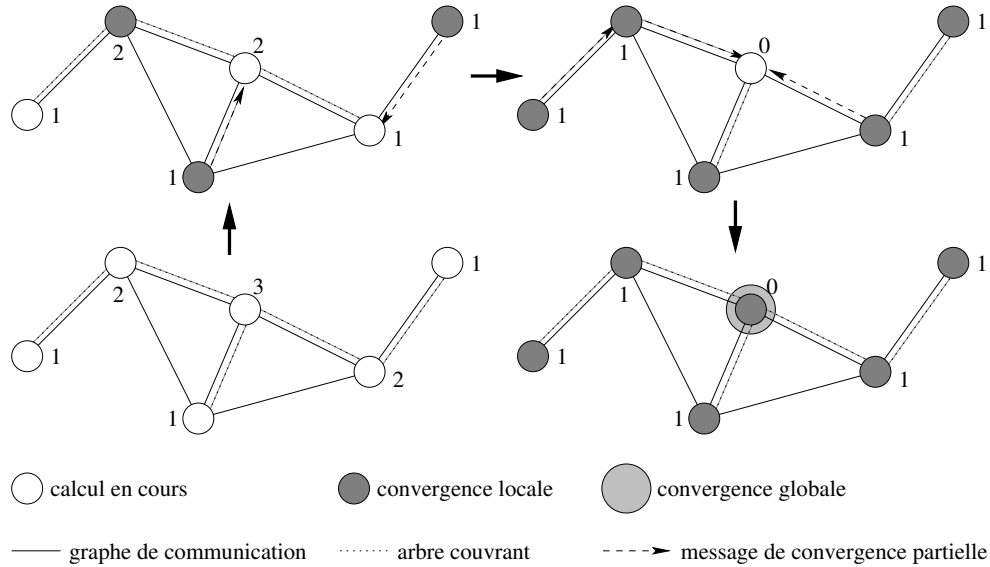


FIG. 2.17 – Processus de détection de la convergence globale basé sur l'élection de leader. Les chiffres indiquent le nombre de voisins dans l'arbre couvrant dont le processeur n'a pas encore reçu de message de convergence partielle.

Les fonctions de réception des messages de convergence (partielle ou globale) sont exécutées dans des processus légers. La fonction de réception des messages de convergence partielle consiste à décrémenter la valeur de `NbMsgNonRec` et à mémoriser l'expéditeur du message. La fonction de réception de la convergence globale positionne la valeur de `ConvGlobale` à Vrai.

Algorithme 15 Détection décentralisée de la convergence globale

... variables nécessaires au calcul ...

NbMsgNonRec = *Nombre de voisins dans l'arbre couvrant dont le processeur courant n'a pas encore reçu de message de convergence partielle, initialisé au nombre de voisins du processeur dans cet arbre*

NbItSousSeuil = *Nombre d'itérations successives sous le seuil de convergence, initialisé à 0*

SousSeuil = *Booléen indiquant que l'itération effectuée a un résidu sous le seuil, initialisé à Faux*

ConvLocale = *Booléen indiquant que le processeur a atteint la convergence locale, initialisé à Faux*

ConvGlobale = *Booléen indiquant que la convergence globale est atteinte, initialisé à Faux*

... Initialisations ...

répéter

si ConvLocale = Faux **alors**

 ... Calculs et communications du processus itératif ...

 Évaluation de SousSeuil

si SousSeuil = Vrai **alors**

 NbItSousSeuil = NbItSousSeuil + 1

si NbItSousSeuil = NbItCvLoc **alors**

 ConvLocale = Vrai

fin si

sinon

 NbItSousSeuil = 0

fin si

sinon

si NbMsgNonRec = 0 **alors**

 ConvGlobale = Vrai

sinon

si NbMsgNonRec = 1 **alors**

 Envoi d'un message de convergence partielle au seul voisin dont on n'a pas reçu un tel message

fin si

fin si

fin si

jusqu'à ConvGlobale = Vrai

Envoi d'un message de convergence globale à tous les voisins dont on n'a pas déjà reçu un message de convergence globale

... Post-traitements ...

La preuve du bon fonctionnement de cet algorithme est donnée dans [8].

2.5.2 Version pratique

La difficulté principale que l'on rencontre lors de la mise en œuvre de l'algorithme 15 vient de la détermination de la valeur de **NbItCvLoc**. En effet, comme nous l'avons vu dans la partie 2.3.1, bien que cette valeur soit définie théoriquement, il n'est pas possible en pratique de l'évaluer de manière précise.

Or, cette valeur joue un rôle central pour le bon fonctionnement et l'efficacité de l'algorithme de détection. Si elle est fixée au-dessous de sa valeur théorique réelle, de fausses convergences locales peuvent être détectées menant à une détection erronée de la convergence globale. Si, au contraire, cette valeur est fixée au-dessus de sa valeur théorique réelle, alors les convergences locales sont correctement détectées ainsi que la convergence globale, mais avec un délai supplémentaire qui réduit l'efficacité de l'algorithme.

Ainsi, même si l'Algorithme 15 est valide d'un point de vue théorique, il n'est pas directement utilisable en pratique sous cette forme. Pour contourner ce problème, nous sommes amenés à utiliser un mécanisme de détection de la convergence locale similaire à celui de l'Algorithme 8. Ce qui implique aussi l'utilisation de messages d'annulation des convergences partielles.

La version pratique est présentée dans l'Algorithme 16. Elle reprend tous les mécanismes de l'Algorithme 8 en y incluant l'algorithme d'élection de leader. Ainsi, la constante théorique **NbItCvLoc** est remplacée par **NbItCvLocSup** pour détecter les convergences locales supposées et **ConvLocaleSup** remplace **ConvLocale**. Contrairement à la version théorique, après une telle détection, il est possible que le résidu repasse au-dessus du seuil. Dans ce cas, la convergence locale supposée est annulée sur le processeur et si un message de convergence partielle avait été envoyé précédemment, alors un message d'annulation est envoyé à la même destination. Le processeur qui reçoit un tel message d'annulation incrémente sa variable locale **NbMsgNonRec** et annule la convergence partielle de l'expéditeur. Si cette annulation arrive après que ce processeur a lui-même envoyé un message de convergence partielle à un de ses voisins, alors il envoie aussi une annulation à ce voisin. Cela permet de propager l'annulation de la convergence locale d'un processeur dans le graphe et d'éviter une fausse détection de la convergence globale. À la différence de l'Algorithme 8 où le temps d'attente pris en compte après la détection de la convergence globale est le temps maximal de communication entre un processeur et le maître, ce temps d'attente doit être ici le temps maximal de propagation d'une annulation sur l'ensemble du graphe. La valeur **TpsMaxProp** remplace donc **TpsMaxAnnul**. Les fonctions de réception des messages de convergence partielle et d'annulation de ces convergences sont respectivement décrites dans les Algorithmes 17 et 18. La fonction de réception de la convergence globale est identique à la version précédente.

On peut remarquer que si **NbItCvLocSup** est supérieur ou égale à la valeur théorique, on revient au même comportement que dans l'Algorithme 15 et la convergence globale est

Algorithme 16 Version pratique de l'Algorithme 15

... variables de l'Algorithme 15 sauf ConvLocale ...
ConvLocaleSup = *Booléen indiquant que le processeur a atteint une convergence locale supposée, initialisé à Faux*
... Initialisations ...
répéter
... Calculs et communications du processus itératif ...
Évaluation de SousSeuil
si ConvLocaleSup = Faux **alors**
 si SousSeuil = Vrai **alors**
 NbItSousSeuil = NbItSousSeuil + 1
 si NbItSousSeuil = NbItCvLocSup **alors**
 ConvLocaleSup = Vrai
 fin si
 sinon
 NbItSousSeuil = 0
 fin si
sinon
 si SousSeuil = Faux **alors**
 ConvLocaleSup = Faux
 ConvGlobale = Faux
 NbItSousSeuil = 0
 si un message de convergence partielle a déjà été envoyé **alors**
 Envoi d'un message d'annulation avec le numéro d'itération courante au même destinataire que ce précédent message
 fin si
sinon
 si NbMsgNonRec = 0 **alors**
 ConvGlobale = Vrai
 sinon
 si NbMsgNonRec = 1 **alors**
 Envoi d'un message de convergence partielle avec le numéro d'itération courante au seul voisin dont on n'a pas reçu un tel message
 fin si
 fin si
fin si
jusqu'à intervalle de temps où ConvGlobale = Vrai > TpsMaxProp
Envoi d'un message de convergence globale à tous les voisins dont on n'a pas déjà reçu un message de convergence globale
... Post-traitements ...

Algorithme 17 Fonction RecConvPart()

```

NumSource = Numéro du processeur expéditeur du message
NumIter = Itération à laquelle le message a été envoyé par l'expéditeur

si NumIter > NumIterMaxAn[NumSource]
    et NumIterMaxAn[NumSource] > NumIterMaxCv[NumSource] alors
        NbMsgNonRec = NbMsgNonRec - 1
    fin si
si NumIter > NumIterMaxCv[NumSource] alors
    NumIterMaxCv[NumSource] = NumIter
fin si

```

Algorithme 18 Fonction RecAnnulConvPart()

```

NumSource = Numéro du processeur expéditeur du message
NumIter = Itération à laquelle le message a été envoyé par l'expéditeur

si NumIter > NumIterMaxCv[NumSource]
    et NumIterMaxCv[NumSource] > NumIterMaxAn[NumSource] alors
        NbMsgNonRec = NbMsgNonRec + 1
        ConvGlobale = Faux
    fin si
si NumIter > NumIterMaxAn[NumSource] alors
    NumIterMaxAn[NumSource] = NumIter
fin si

```

correctement détectée sans qu'aucun message d'annulation ne soit généré. Néanmoins, en pratique cette valeur ne sera pas choisie très grande et il y a donc peu de chances qu'elle atteigne la valeur théorique.

Finalement, ce mécanisme nous assure bien que lorsque la convergence globale est détectée, tous les processeurs sont en convergence locale. Ce qui correspond au même critère d'arrêt utilisé dans les versions synchrone et séquentielle. Par contre, cette version pratique possède un temps d'attente après détection de la convergence globale différent de celui de la version centralisée. De plus, comme mentionné au début de cette partie, la technique de transmission (forwarding) permet de conserver l'algorithme centralisé sur un graphe non complet de processeurs. Son principe consiste à effectuer un routage explicite des messages des processeurs vers le maître en effectuant uniquement des communications entre voisins. Cependant, cette technique implique de disposer sur chaque processeur d'une connaissance au moins partielle du graphe d'interconnexion des processeurs pour pouvoir effectuer les relais convenablement. Il est donc intéressant de comparer expérimentalement les performances globales des ces trois algorithmes dans les contextes de grappe et méta-grappe.

Toutefois, une première contrainte pour effectuer ces tests vient de l'utilisation de la version centralisée qui implique que les contextes choisis ne doivent pas avoir de restrictions

d'accès entre processeurs. Une seconde contrainte vient du fait qu'il n'y a pas de moyen direct d'évaluer uniquement la partie de détection de convergence car l'instant précis auquel commence ce processus de détection ne peut être identifié. Ainsi, pour évaluer les efficacités relatives de ces trois algorithmes de détection, sont comparés les temps d'exécution globaux d'un même algorithme AIAC utilisant tour à tour une des trois variantes de détection.

Le problème utilisé pour réaliser ces expérimentations est le problème d'advection-diffusion à deux dimensions qui modélise l'évolution des concentrations de deux espèces chimiques dans un espace bidimensionnel. Ce choix est motivé par deux raisons. D'une part, le souhait de montrer que les algorithmes AIAC ne sont pas limités aux problèmes à une dimension. D'autre part, le fait que la méthode de résolution utilisée, décrite ci-dessous, fasse intervenir une séquence de processus itératifs séparés par des synchronisations plutôt qu'un seul processus itératif, augmente l'impact de l'algorithme de détection de convergence sur les performances globales et permet donc de mieux mettre en évidence les différences éventuelles entre les trois algorithmes de détection de convergence testés.

Le problème est résolu en discrétisant l'espace sur une grille bidimensionnelle (x, z) . Les évolutions des concentrations sur un intervalle de temps fixé sont données en chaque point (x, z) par les équations différentielles suivantes :

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial z} K_v(z) \frac{\partial c^i}{\partial z} + R^i(c^1, c^2, t) \quad (2.18)$$

où $i = 1, 2$ dénote le numéro de l'espèce chimique et

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2 \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 + q_4(t) c^2 \end{aligned} \quad (2.19)$$

avec

$$\begin{aligned} K_h &= 4,0 \times 10^{-6} & V &= 10^{-3} & q_1 &= 1,63 \times 10^{-16} \\ K_v(z) &= 10^{-8} e^{\frac{z}{5}} & c^3 &= 3,7 \times 10^{16} & q_2 &= 4,66 \times 10^{-16} \\ q_j(t) &= e^{-a_j / \sin(\omega t)} & & \text{pour } \sin(\omega t) > 0 \\ q_j(t) &= 0 & & \text{sinon} \end{aligned}$$

et $j = 3, 4$, $\omega = \pi/43200$, $a_3 = 22,62$ et $a_4 = 7,601$.

L'intervalle de temps est de 7200 secondes et les conditions initiales sont les suivantes :

$$\begin{aligned} c^1(x, z, 0) &= 10^6 \alpha(x) \beta(z) \\ c^2(x, z, 0) &= 10^{12} \alpha(x) \beta(z) \end{aligned} \quad (2.20)$$

avec

$$\begin{aligned} \alpha(x) &= 1 - (0,1x - 1)^2 + (0,1x - 1)^4 / 2 \\ \beta(z) &= 1 - (0,1z - 1)^2 + (0,1z - 4)^4 / 2 \end{aligned} \quad (2.21)$$

La discrétisation le long de x et z permet de récrire ce système d'EDPs sous la forme d'un système d'EDOs :

$$\frac{dy(t)}{dt} = f(y(t), t) \quad \text{avec} \quad y = (c^1, c^2) \quad (2.22)$$

Cette dernière équation est résolue en utilisant la technique des différences finies. La méthode implicite d'Euler est utilisée pour effectuer la discrétisation sur l'intervalle de temps, ce qui donne :

$$y(t+h) - y(t) = h.f(y(t+h), t+h) \quad (2.23)$$

et en posant

$$G(t) = y(t+h) - y(t) - h.f(y(t+h), t+h)$$

la solution de l'équation 2.23 est obtenue en résolvant $G(t) = 0$ par la méthode itérative de Newton. Chaque étape du processus de Newton requiert la résolution d'un système linéaire.

Ainsi, la solution globale est calculée par une boucle principale sur les pas de temps et chaque pas de temps est calculé par deux processus itératifs imbriqués. Le processus le plus général correspond aux étapes de la méthode de Newton et le processus interne correspond à la résolution du système linéaire obtenu à chacune de ces étapes. Dans un contexte de calcul parallèle, il est nécessaire de synchroniser les processeurs entre chaque pas de temps. En effet, pour commencer le calcul d'un pas de temps, toutes les concentrations au pas de temps précédent doivent être connues. Par contre, à l'intérieur d'un pas de temps, les calculs peuvent être réalisés de manière asynchrone.

En fait, il y a deux possibilités de mise en œuvre de la méthode de Newton. La première consiste à l'appliquer sur le système complet et à utiliser un solveur linéaire parallèle asynchrone sur ce système. Néanmoins, des synchronisations sont malgré tout nécessaires entre deux itérations successives du processus de Newton. La seconde approche, appelée Newton multi-décomposition (multisplitting Newton en anglais), consiste à décomposer le système en plusieurs sous-systèmes qui sont résolus localement par un solveur linéaire séquentiel en tenant compte des dépendances entre les sous-systèmes. Cette méthode présente l'avantage de ne requérir qu'une seule synchronisation entre chaque pas de temps (voir [53] pour une description détaillée de cette méthode). Dans les expérimentations suivantes, c'est cette méthode qui a été choisie avec, pour solveur linéaire séquentiel, la méthode GMRES [127]. Ainsi, la grille est répartie sur les processeurs en la découpant en bandes horizontales. Les concentrations en un point (x, z) dépendant aussi de celles en $(x, z-1)$, $(x, z+1)$, $(x-1, y)$ et $(x+1, y)$, on en déduit que les dépendances pour le calcul d'une bande horizontale sont la bande au-dessus et au-dessous de celle-ci. Les processeurs sont donc logiquement organisés en topologie linéaire.

La mise en œuvre a été réalisée avec l'environnement OmniOrb 4 [93] qui suit la version 2.1 de la norme Corba [120]. Cette norme établit une architecture logicielle et des mécanismes pour permettre à différentes applications de communiquer dans des environnements hétérogènes. OmniOrb en est une mise en œuvre particulièrement robuste et performante. Les caractéristiques les plus intéressantes pour la mise en œuvre des algorithmes AIAC sont le support des processus légers et le mécanisme d'appel de procédure à distance (Remote Processing Call en anglais) qui permettent de réaliser efficacement les communications asynchrones. De plus, il bénéficie d'une grande souplesse de déploiement sur les grappes et méta-grappes.

Comme indiqué précédemment, deux contextes expérimentaux sont utilisés pour la comparaison des algorithmes. Le premier est une grappe locale constituée de quinze Duron 800Mhz, dix Pentium IV 1,7Ghz et quinze Pentium IV 2,4Ghz reliés par un réseau Ethernet à 100Mb/s. Le second est une méta-grappe comportant un Pentium III 833Mhz, un Pentium III 900Mhz, un Pentium IV 1,7Ghz, trois Athlon XP 2000, deux Pentium IV 2,4Ghz, un Pentium IV 2,6Ghz et un Athlon XP 2800 répartis sur quatre sites reliés entre eux par des liens Ethernet à 10Mb/s et un lien ADSL à 128Kb/s. Ces deux ensembles de machines ont été utilisés sans aucune charge supplémentaire pendant les tests.

Enfin, deux discrétisations différentes de l'espace bidimensionnel ont été utilisées sur la méta-grappe, une grille de 400×400 points et une de 600×600 . Comme la grappe locale comporte beaucoup plus de machines, une grille de 800×800 points a été utilisée pour conserver une quantité de calcul représentative.

Les résultats présentés dans la Table 2.6 sont une moyenne de dix exécutions.

contexte et taille	centralisé pur	centralisé par transmission	décentralisé
méta-grappe (400×400)	163	162	155
méta-grappe (600×600)	649	651	639
grappe (800×800)	368	373	373

TAB. 2.6 – Temps d'exécution (en secondes) de l'algorithme AIAC de résolution du problème d'advection-diffusion 2D avec trois variantes de détection de la convergence globale.

Dans le contexte de la grappe locale, il n'est pas très surprenant que la version centralisée obtienne les meilleures performances puisque les communications vers le processeur maître sont très rapides. Par contre, l'algorithme décentralisé obtient les meilleures performances dès que l'on passe dans le contexte de la méta-grappe. Dans ce contexte, le résultat le plus intéressant est que la version centralisée par transmission est moins performante que la version décentralisée. Cela peut s'expliquer par le fait que, en plus des gestions différentes des messages de convergence, la version décentralisée ne force pas la détection de la convergence sur un processeur précis. Ce résultat est important car dans la plupart des méta-grappes, seules ces deux versions seront utilisables. De plus, un écart encore plus net entre les performances de ces deux versions peut logiquement être escompté dans des contextes de méta-grappes à plus grande échelle, comportant plus de machines et réparties sur plus de sites.

Ainsi, l'algorithme décentralisé de détection de la convergence globale n'est pas seulement intéressant au niveau du déploiement des algorithmes asynchrones mais contribue aussi à améliorer les performances.

2.6 Environnements de développement

Dans les parties précédentes de ce chapitre, nous avons vu que différents environnements de développement peuvent être utilisés pour la mise en œuvre des algorithmes asynchrones.

Néanmoins, ils présentent des caractéristiques qui peuvent être assez différentes. Il est donc intéressant de comparer expérimentalement ces environnements en fonction des performances qu'ils obtiennent et des facilités de déploiement et de programmation qu'ils offrent. Ma contribution à ce sujet, détaillée dans les études [17] et [4] et présentée par la suite, n'a pas seulement consisté à effectuer une telle comparaison mais a aussi permis de dégager un certain nombre de caractéristiques qui sont essentielles à tout environnement de développement visant à offrir une mise en œuvre et un déploiement efficace des algorithmes asynchrones.

Après une description des différents environnements testés, une comparaison est effectuée avec deux types communs de problèmes scientifiques en analysant les performances obtenues mais aussi les facilités de programmation et de déploiement. Enfin, les caractéristiques nécessaires aux algorithmes asynchrones sont précisées.

2.6.1 Environnements testés

Les algorithmes présentés précédemment montrent clairement que la première caractéristique nécessaire à un environnement de développement est la gestion des processus légers. Les bibliothèques standard mono-processus telles que MPI ou PVM n'incluent pas cette fonctionnalité et requièrent un placement explicite des communications dans l'algorithme principal. Ainsi, bien que la programmation des algorithmes asynchrones soit possible avec ces bibliothèques, elle est beaucoup moins souple et efficace. Ces environnements mono-processus n'entrent donc pas dans cette comparaison. De plus, les algorithmes asynchrones étant particulièrement efficaces dans les contextes de méta-grappes de calcul, il est nécessaire de choisir des environnements qui permettent leur mise en œuvre dans ces contextes. Ainsi, trois environnements relativement connus dans la communauté du parallélisme et présentant des approches conceptuelles différentes ont donc été sélectionnés, il s'agit de PM2 [118], MPICH/Madeleine [49] et OmniOrb 4 [93].

L'environnement PM2 a été développé pour permettre la mise en œuvre efficace d'applications parallèles irrégulières sur des architectures distribuées. Il permet de gérer un grand nombre de processus légers et possède même des mécanismes de migration. Son organisation interne est basée sur deux composants logiciels : Marcel et Madeleine. Marcel est un gestionnaire de processus compatible POSIX et Madeleine est une interface de communication générique qui peut être utilisée avec différents protocoles tels que VIA, BIP, SBP, SCI, MPI, PVM et TCP. C'est ce dernier qui a été utilisé dans les tests suivants.

L'environnement MPICH/Madeleine est une version de MPI qui utilise aussi Marcel. Il offre donc la possibilité de gérer des processus légers dans l'environnement MPI qui est un des plus utilisés dans la communauté du parallélisme. Bien qu'il utilise le même gestionnaire de processus que PM2, leur comparaison est pertinente car ils utilisent des schémas de communication différents (explicite dans MPI et par RPC dans PM2).

Enfin, OmniOrb 4 est un environnement robuste et efficace qui suit la norme Corba 2.1. Il a son propre gestionnaire de processus et utilise un schéma de communication orienté objet

basé sur le RPC. Il peut paraître surprenant d'utiliser un environnement Corba pour mettre en œuvre des algorithmes parallèles. Néanmoins, même si ce type d'environnement n'a pas été initialement conçu pour ce genre d'utilisation, il présente toutes les fonctionnalités nécessaires. Il est donc intéressant de le comparer aux deux environnements précédents.

Il existe quelques autres environnements présentant les fonctionnalités requises mais ils ne sont pas conçus dans un objectif de performance. Par exemple, l'environnement Jace [51] a été spécialement conçu pour la mise en œuvre des algorithmes asynchrones. Néanmoins, l'objectif des auteurs étant la portabilité, il a été écrit en Java [48] et ne peut donc pas rivaliser, en termes de performances, avec les autres environnements.

2.6.2 Performances

Deux problèmes ont été utilisés pour effectuer cette comparaison, le problème linéaire creux présenté dans la partie 2.3.2 et le problème d'advection-diffusion 2D présenté dans la partie 2.5.2.

Pour obtenir une comparaison pertinente des environnements, une attention particulière a été portée sur l'utilisation du même schéma algorithmique général avec les trois environnements. Pour chaque problème, le même schéma de calculs, le même schéma de communication, le même algorithme de détection de la convergence globale et la même procédure d'arrêt ont été mis en œuvre d'un environnement à l'autre et les mêmes paramètres ont été utilisés. Cependant, pour mettre en évidence les éventuelles différences d'adaptation aux algorithmes asynchrones entre les environnements, leurs atouts techniques respectifs ont été exploités au mieux. Ainsi, dans chaque cas, la mise en œuvre la plus efficace respectant les schémas communs a été retenue. En fait, la différence principale se situe au niveau de la gestion des processus légers utilisés pour les communications.

Au départ, une tentative a été faite d'utiliser exactement le même schéma avec tous les environnements qui consistait à créer autant de processus légers que nécessaire pour les envois et les réceptions. Malheureusement, cela posait des problèmes avec PM2 et MPICH/Madeleine. Ainsi, après une série de tests préliminaires, les stratégies opérationnelles les plus efficaces, présentées dans la Table 2.7 ont été utilisées.

		Problème linéaire creux	Problème non linéaire
PM2	envois	un processus	deux processus
	réceptions	processus créés à la demande	un processus
MPICH/ Madeleine	envois	un processus	deux processus
	réceptions	un processus	deux processus
OmniOrb 4	envois	N processus	deux processus
	réceptions	processus créés à la demande	processus créés à la demande

TAB. 2.7 – Gestion des processus selon le problème traité et l'environnement utilisé (N est le nombre de processeurs).

Ces différences peuvent avoir une influence sur les performances finales car la vitesse de convergence sur chaque processeur dépend en partie de la fréquence des mises à jour des dépendances. Si à un instant donné, les envois ou les réceptions sont retardés à cause de l'environnement de programmation, alors le processus itératif global peut être ralenti. D'un autre côté, l'utilisation d'un processus léger par communication peut entraîner un surcoût lorsque le nombre de processeurs voisins est très important.

Problème linéaire

Un exemple classique de méta-grappe répartie sur trois sites reliés par des connexions Ethernet à 10Mb/s a été utilisée pour traiter ce problème. Pour assurer la convergence du processus itératif, la matrice a été choisie avec un rayon spectral inférieur à un. Elle a une taille de 2000000×2000000 avec une répartition des valeurs non nulles sur trente sous-diagonales.

Les résultats obtenus, moyennes de dix exécutions, sont présentés dans la Table 2.8. La version synchrone, développée sous MPI, a aussi été incluse pour fournir une base de référence. On peut noter sans surprise que les versions asynchrones sont toutes plus rapides que la version synchrone. Par contre, il y a des différences sensibles de performances entre les versions asynchrones. En effet, on constate que OmniOrb et PM2 sont relativement proches mais MPICH est nettement en retrait puisqu'il nécessite 32% de temps en plus.

version	temps d'exécution (s)	ratio
synchrone MPI	914	1
asynchrone PM2	551	1,66
asynchrone MPICH/Madeleine	672	1,36
asynchrone OmniOrb 4	507	1,80

TAB. 2.8 – Temps d'exécution (en secondes) de l'algorithme de résolution du problème linéaire creux sous différents environnements.

En fait, le ratio élevé des communications par rapport aux calculs dans ce problème (cf partie 2.3.2) fait ressortir plus particulièrement les différences d'efficacité de la gestion interne des communications entre les environnements.

Problème non linéaire

Pour le problème non linéaire, deux contextes de calcul supplémentaires ont été utilisés. D'une part, une autre méta-grappe représentant un contexte plus difficile puisque répartie sur quatre sites avec certains liens ADSL à 512Kb/s en réception et 128Kb/s en envoi. Et d'autre part, une grappe locale hétérogène composée de Durons 800Mhz, de Pentiums IV 1,7Ghz et de Pentiums IV 2,4Ghz. Les paramètres utilisés sont une grille de discrétisation de 600×600 points, un intervalle de temps de 2160 secondes par pas de 180 secondes. Enfin, comme pour le problème linéaire, les résultats suivants sont des moyennes de dix exécutions.

La Table 2.9 présente les résultats obtenus sur les deux méta-grappes. Là aussi, la version synchrone développée avec MPI sert de référence.

méta-grappe	version	temps d'exécution (s)	ratio
Ethernet	synchrone MPI	2510	1
	asynchrone PM2	563	4,46
	asynchrone MPICH/Madeleine	565	4,44
	asynchrone OmniOrb 4	595	4,22
Ethernet et ADSL	synchrone MPI	3042	1
	asynchrone PM2	612	4,97
	asynchrone MPICH/Madeleine	605	5,03
	asynchrone OmniOrb 4	664	4,58

TAB. 2.9 – Temps d'exécution (en secondes) de l'algorithme de résolution du problème d'advection-diffusion sous différents environnements.

On constate que pour ce problème, les rôles sont inversés entre OmniOrb 4 et MPICH. Toutefois, les différences sont moins importantes puisque OmniOrb 4 obtient des temps entre 5 et 10% plus lents que MPICH. Il faut noter que dans ce problème, la charge des communications est bien moins importante que dans le problème précédent puisque les processeurs échangent moins de données vers moins de voisins.

Avec la grappe locale, les environnements sont testés dans un contexte de machines hétérogènes reliées par un réseau homogène et relativement rapide (100Mb/s). Le nombre de machines disponibles étant plus important, il a été possible de le faire varier en conservant la même proportion de chaque type de machine et en les intercalant dans l'organisation logique pour respecter le passage à l'échelle. Les évolutions des temps d'exécution avec chaque environnement pour une taille de problème de 1000×1000 sont représentées dans la Figure 2.18.

Dans ce contexte, on retrouve des résultats similaires aux précédents avec les trois versions asynchrones plus rapides que la version synchrone, PM2 et MPICH quasiment équivalents et OmniOrb légèrement en retrait. Cela n'est pas très surprenant car PM2 et MPICH ont été conçus pour fonctionner sur des grappes locales alors que OmniOrb est prévu au départ pour des communications client/serveur distantes. Leurs systèmes internes de communication et de gestion des processus ne sont donc pas conçus pour optimiser les mêmes aspects et OmniOrb se retrouve désavantagé dans ce cas. Le fait que les courbes semblent se rejoindre vers le plus grand nombre de machines vient simplement du fait qu'on se rapproche de la limite du parallélisme pour cette taille de problème. Par contre, un point intéressant est que pour la résolution d'un problème donné, l'efficacité maximale du parallélisme sera obtenue avec moins de processeurs en asynchrone qu'en synchrone. Ainsi, l'asynchronisme permet aussi de réaliser une économie de moyens.

Finalement, ces différentes expériences montrent que OmniOrb semble plus adapté aux problèmes faisant intervenir des volumes de communication importants alors que MPICH/Madeleine et PM2 restent relativement proches l'un de l'autre. PM2 paraît toutefois

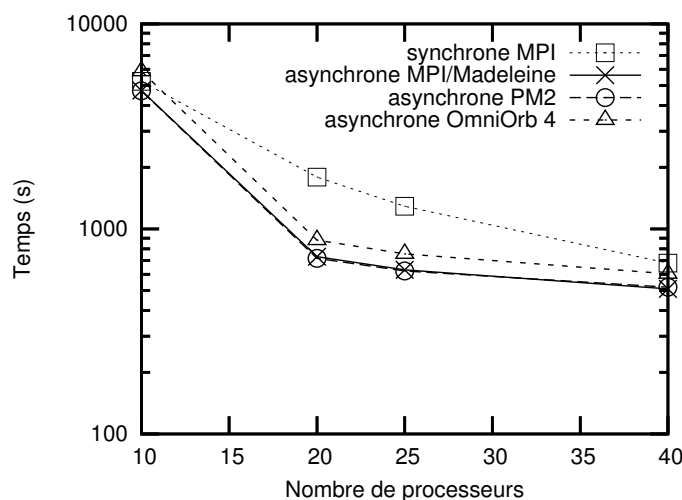


FIG. 2.18 – Temps d'exécution de l'algorithme AIAC de résolution du problème d'advection-diffusion 2D sur une grappe locale hétérogène en fonction du nombre de processeurs pour chaque environnement.

plus polyvalent que les deux autres puisqu'il n'obtient jamais les moins bonnes performances. Cependant, les trois environnements testés obtiennent des performances globalement similaires et d'autres critères comme les facilités de développement et de déploiement peuvent aussi être pris en compte pour effectuer une mise en œuvre.

2.6.3 Programmation

La facilité de programmation est un critère assez subjectif qui dépend beaucoup du programmeur et de son objectif. Cette comparaison se focalise donc sur les fonctionnalités spécifiquement utilisées pour la mise en œuvre des algorithmes asynchrones.

Concernant MPICH/Madeleine, le recours à la bibliothèque Marcel permet de gérer assez facilement l'asynchronisme avec un nombre réduit de mutex et permet l'utilisation des fonctions classiques d'envoi et réception de MPI. Cela présente un avantage important car l'interface de programmation de MPI est simple et familière à tout développeur de programmes parallèles. Par contre, certains aspects ne sont pas pris en compte tels que l'hétérogénéité des machines. Il est donc, par exemple, nécessaire de gérer manuellement les conversions de données transmises entre deux machines n'ayant pas les mêmes représentations numériques. Ce problème se retrouve dans PM2 mais pas dans OmniOrb 4 qui a été prévu pour une utilisation entre machines différentes et gère ces différences de manière transparente.

Concernant PM2 et OmniOrb 4, ils sont relativement similaires en ce qui concerne les communications puisqu'ils utilisent des RPCs. Dans PM2, les données à envoyer doivent être empaquetées avant l'appel de procédure distante alors que dans OmniOrb les données sont passées en arguments de la fonction appelée.

Certaines spécificités d'OmniOrb lui valent quelques difficultés supplémentaires. Notamment, le recours à un langage de déclaration d'interface (IDL en anglais) pour générer le code initial implique une étape supplémentaire dans le développement. Enfin, au moment de l'initialisation du système de communication, il est nécessaire d'établir les liens entre voisins de manière plus ou moins manuelle. Cependant, cette phase d'initialisation est assez générique et peut donc être réutilisée d'un programme à l'autre.

Au niveau de la configuration des environnements, celle de MPICH est relativement simple puisqu'elle correspond globalement à une liste de machines dans un fichier texte. Celle de PM2 est un peu plus complexe car il est possible de spécifier différents sous-réseaux et leurs compositions. Pour OmniOrb, chaque site doit être configuré pour localiser ce que l'on appelle le service de nommage qui sert à rediriger les appels de procédures aux bons endroits. Ce service doit être localisé sur une machine accessible depuis tous les sites de la méta-grappe.

Ainsi, parmi les trois environnements testés, MPICH/Madeleine semble être le plus simple à utiliser devant PM2 et OmniOrb.

2.6.4 Déploiement

Concernant le déploiement des algorithmes asynchrones, les difficultés majeures se situent au niveau de l'hétérogénéité des machines utilisées et des contraintes de sécurité entre ces machines qui impliquent souvent une visibilité partielle du système complet, notamment dans les méta-grappes. Il est donc important de disposer d'un environnement permettant une mise en œuvre dans ces contextes particuliers.

En ce qui concerne OmniOrb 4, il présente une grande flexibilité dans ce domaine. Son orientation client/serveur multi-plateforme permet de gérer l'hétérogénéité des machines de manière transparente pour le programmeur et de constituer dynamiquement un graphe de communication non complet entre ces machines, ce qui est essentiel pour pouvoir utiliser une méta-grappe ayant des restrictions d'accès entre ses sites. On voit ici que ce qui est un inconvénient au niveau de la facilité de programmation peut devenir un avantage au niveau du déploiement.

Les environnements PM2 et MPICH sont assez similaires et présentent, quant à eux, une restriction importante puisqu'ils nécessitent un graphe de connexion complet entre les machines du système. De plus, leur portabilité est plus réduite puisqu'ils ne supportent pas complètement l'utilisation de plusieurs types de machines et/ou systèmes d'exploitation dans une même grappe ou méta-grappe. Par contre, ils autorisent l'utilisation de différents protocoles de communications dans la méta-grappe, ce qui est utile lorsque les réseaux locaux des différents sites utilisés ne sont pas les mêmes.

Finalement, c'est OmniOrb qui offre la plus grande flexibilité de déploiement. Cela n'est pas une grande surprise car c'est le seul des trois environnements qui est prévu au départ pour faire communiquer des applications distantes.

2.6.5 Caractéristiques de développement des AIAC

Cette étude comparative a permis de mettre en évidence les éléments nécessaires à une mise en œuvre efficace des algorithmes asynchrones.

L'environnement de développement doit évidemment posséder un système de communication point à point avec, si possible, des versions bloquantes. Pour permettre un déploiement efficace, il doit supporter plusieurs protocoles de communication et surtout autoriser l'utilisation d'un graphe d'interconnexion non complet entre les machines. Ces deux dernières caractéristiques ne sont généralement pas incluses dans la plupart des environnements de développement parallèle.

L'environnement doit aussi gérer les processus légers de manière équitable. Cela est très important lors de la gestion de plusieurs processus d'envois/réceptions sur un même processeur. En fait, si l'ordonnanceur des processus légers n'est pas équitable, il est possible d'avoir toujours les mêmes processus qui sont actifs et toujours les mêmes qui sont inactifs. Ainsi, les communications gérées dans ces derniers ne sont pas effectuées. De plus, lorsqu'un système de RPC est utilisé, il est important que les réceptions puissent être effectuées dans des processus activés sur demande. Des tests annexes ont révélé que la gestion des processus n'était pas toujours équitable sur certains environnements.

Enfin, en complément des processus légers, un système de gestion des mutex est aussi nécessaire pour gérer de manière cohérente les différentes communications et accès aux données locales, notamment avec l'algorithme d'équilibrage.

Lorsque toutes ces caractéristiques sont réunies, on peut s'attendre à des performances optimales quel que soit le contexte d'utilisation. Cependant, il est aussi intéressant de voir que les algorithmes asynchrones peuvent être mis en œuvre avec des environnements moins complets tout en présentant de bonnes performances. Cela est important pour la diffusion des algorithmes asynchrones dans la communauté scientifique puisque chacun peut les utiliser avec l'environnement de développement qui lui est le plus familier.

2.7 Conclusion et perspectives

L'ensemble des travaux présentés dans ce chapitre sur les algorithmes asynchrones dans le cadre continu montre que ceux-ci sont bien plus performants que leurs homologues synchrones. Leurs avantages par rapport à ces derniers sont nombreux et variés. Le premier et le plus évident est un recouvrement efficace et implicite des communications par des calculs qui leur apporte une grande robustesse aux variations dynamiques du système de calcul. Ils permettent d'exploiter efficacement des systèmes comportant des machines et des liens de communication hétérogènes. De plus, ils autorisent l'utilisation de techniques d'optimisation complémentaires telles que l'équilibrage de charge et offrent aussi une tolérance aux pertes des messages relatifs aux données de calcul.

Ainsi, ces algorithmes asynchrones ne représentent pas simplement une adaptation logicielle à un nouveau type de système de calcul mais correspondent à une véritable évolution algorithmique. Ils permettent non seulement d'exploiter efficacement les nouveaux systèmes de calcul apportés par les progrès récents sur les réseaux de communication mais se montrent aussi plus efficaces que les algorithmes classiquement utilisés sur les systèmes plus anciens. Ils démontrent ainsi leur supériorité algorithmique.

Si les travaux présentés ici démontrent l'intérêt essentiel des algorithmes asynchrones, nous avons vu que cette évolution logicielle a pourtant commencé au niveau théorique il y a plusieurs décennies. Cependant, leur exploitation pratique a longtemps été limitée par un manque d'intérêt de la part de la communauté scientifique, sans doute lui-même suscité par un manque d'environnements de développement permettant leur mise en œuvre efficace.

L'objectif de ma démarche a donc été de mettre en lumière tout l'intérêt de l'asynchronisme dans les algorithmes itératifs parallèles pour résoudre des problèmes scientifiques linéaires et non linéaires dans le domaine continu. Il reste cependant encore plusieurs points d'étude à approfondir, notamment sur les techniques de multi-décomposition ou encore sur les techniques d'accélération de la convergence en se fondant sur les progressions des résidus de chaque élément. Enfin, même dans le contexte des problèmes continus, la représentation finie des nombres dans les machines implique, dans une certaine mesure et bien que les ensembles de valeurs soient de grande taille, un traitement discret de ces problèmes. Il est donc aussi important d'étudier le comportement des algorithmes asynchrones dans le cadre discret. C'est précisément le sujet du chapitre suivant.

Systèmes dynamiques discrets asynchrones

Au chapitre précédent, l'intérêt de l'ajout de l'asynchronisme dans les algorithmes itératifs parallèles a été montré dans le cadre continu. Cependant, comme indiqué à la fin de ce précédent chapitre, les représentations numériques utilisées sur les machines actuelles sont binaires et de taille fixe. Cela implique que l'ensemble des valeurs de chaque élément calculé est fini et a une taille de 2^b où b est le nombre de bits utilisés. Ainsi, la résolution d'un problème continu est réalisée en définitive dans un domaine discret. Une étude spécifique de ces systèmes discrets peut donc s'avérer fort utile pour mieux comprendre et éventuellement améliorer le comportement des algorithmes de résolution de problèmes du continu.

Au-delà de ces problèmes de représentation des nombres, il existe aussi des problèmes scientifiques de nature discrète. De plus, le cadre discret est souvent utilisé pour la modélisation et l'étude de systèmes complexes, notamment en automatique et en informatique. Il permet une modélisation et une étude comportementale très précise puisque, contrairement au cadre continu, les phénomènes de convergence asymptotique ne sont pas possibles. Ainsi, si un processus dynamique converge vers un état donné, il atteindra effectivement cet état en un temps fini. En un sens, leur comportement est plus simple. Cependant, les résultats de convergence obtenus dans le cadre continu ne peuvent être utilisés directement dans le cadre discret car certaines notions utilisées, telle que la distance dans \mathbb{R}^n , sont trop restrictives pour des ensembles discrets finis, comme il a été montré par J.Bahi dans [50]. Ainsi, une étude spécifique doit être menée sur les systèmes dynamiques discrets en utilisant des outils adaptés à ce cadre.

La première partie de ce chapitre précise les spécificités induites par la nature discrète du système sur le formalisme général du Chapitre 1 et présente une description précise des propriétés de l'évolution asynchrone suivie des outils indispensables à l'analyse du comportement des systèmes discrets. La partie 3.2 présente un bref état de l'art sur ces systèmes. Une étude de la convergence des systèmes booléens asynchrones basée sur des propriétés locales est détaillée dans la partie 3.3. Cette étude a mené à la conception d'une procédure de test de la convergence pour un point fixe et une initialisation donnés. Enfin, nous verrons dans la partie 3.4 l'intérêt que peuvent présenter les systèmes hybrides mixant synchronisme et asynchronisme.

3.1 Formalisme spécifique

La première implication du cadre discret s'applique bien entendu aux ensembles de valeurs des éléments du système. La notion discrète se traduit généralement par le fait que les E_i sont des sous-ensembles finis de \mathbb{N} .

La seconde implication porte sur le temps qui est lui aussi considéré discret entre les itérations. Ainsi, les itérations sont souvent associées à un bloc atomique d'instructions. Cela réduit le champ des variantes possibles d'asynchronisme vues au Chapitre 2 et basées sur la façon dont sont gérées les communications et les mises à jour des éléments. Dans le cadre discret, l'atomicité des itérations rend les variantes semi-flexibles et flexibles caduques. Cependant, la variante d'asynchronisme total avec communications rigides reste très générale et sera celle utilisée dans la suite de ce chapitre.

Enfin, la nature discrète du système permet une description assez précise de l'influence de l'asynchronisme sur le comportement de celui-ci. La présentation des propriétés de ce mode d'évolution permet de mettre en évidence sa complexité et de fournir un cadre précis aux travaux présentés dans la suite de ce chapitre.

3.1.1 Propriétés de l'évolution asynchrone

Pour présenter les propriétés d'évolution du mode asynchrone, il est nécessaire de préciser tout d'abord la notion de *successeur* d'un état. Si l'on considère l'état x^t du système à l'instant t , le successeur de x^t est simplement l'état du système à l'instant suivant, soit x^{t+1} .

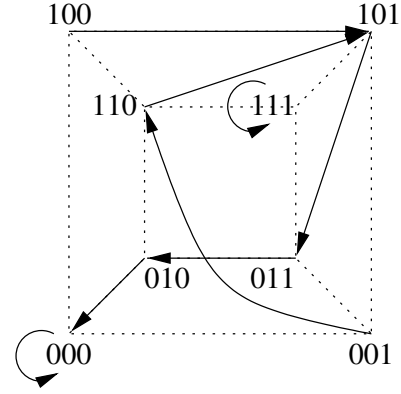
Lorsque l'on étudie précisément les évolutions synchrone et asynchrone, on s'aperçoit que la différence fondamentale entre celles-ci se situe au niveau du nombre de successeurs possibles d'un état donné. En effet, en mode synchrone, à chaque état du système correspond un et un seul successeur. Par contre, ce n'est pas le cas en mode asynchrone et plusieurs successeurs sont possibles selon les éléments qui sont mis à jour et le contexte local de ces mises à jour induit par les retards.

Rappelons ici que le mode totalement asynchrone correspond au mode chaotique avec retards non bornés. Il y a donc en fait deux phénomènes qui interagissent entre eux à prendre en compte. Le premier vient du mode chaotique seul et correspond au fait qu'à chaque itération, certains éléments du système puissent ne pas se mettre à jour. Le second vient des retards entre les éléments et implique pour chacun de ces éléments la prise en compte de l'historique des états suivis par les autres éléments au cours de l'évolution du système et pas uniquement leur état à l'instant courant.

Pour bien comprendre l'influence de chacun de ces phénomènes sur le comportement global du système, ceux-ci sont tout d'abord détaillés séparément dans un exemple simple. L'exemple utilisé consiste en un système de trois éléments à valeurs dans $\{0, 1\}$ chacun et une fonction d'évolution $f = (f_1, f_2, f_3)$. La table des transitions de cette fonction, donnée dans la

Figure 3.1, est accompagnée d'une représentation graphique en permettant une lecture plus intuitive.

x			$f(x)$		
x_1	x_2	x_3	$f_1(x)$	$f_2(x)$	$f_3(x)$
0	0	0	0	0	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1



TAB. 3.1 – Table et représentation graphique de la fonction d'évolution f du système à trois éléments à valeurs dans $\{0, 1\}$.

Concernant le premier point, considérons l'évolution globale du système à partir de l'état 001. En mode synchrone, le successeur de cet état est obligatoirement 110 alors qu'en mode chaotique, il y a plusieurs successeurs possibles (tous dans ce cas) selon les éléments qui sont mis à jour. On constate qu'entre 001 et $f(001) = 110$ les valeurs de tous les éléments changent. Pour chacun des éléments, il y a donc deux valeurs possibles à l'instant suivant selon qu'il est mis à jour ou non. Par exemple, si le premier et le troisième élément sont mis à jour mais pas le second, on passe de l'état 001 à l'état 100. Finalement, deux états possibles pour trois éléments, cela implique $2^3 = 8$ successeurs possibles, qui correspondent à l'ensemble des états du système dans ce cas particulier. Néanmoins, il faut bien voir qu'il n'y a pas toujours autant de successeurs possibles. Si l'on considère l'état 110, son image par f étant 101, on constate que le premier élément conserve la même valeur, qu'il soit mis à jour ou non. Les deux autres éléments changeant de valeur, il y a donc deux valeurs possibles pour deux éléments donc $2^2 = 4$ successeurs possibles qui sont détaillés dans la liste ci-dessous. Dans cette liste, le comportement du premier élément (mise à jour ou non) n'est pas précisé puisqu'il mène à la même valeur :

- 101 : les deux derniers éléments sont mis à jour (équivalent au cas synchrone)
- 100 : le second élément est mis à jour mais pas le troisième
- 111 : le troisième élément est mis à jour mais pas le second
- 110 : aucun des deux derniers éléments n'est mis à jour

On voit donc ici toute la différence entre les itérations synchrones et chaotiques. Les premières sont déterministes alors que les secondes présentent une part d'indéterminisme qui multiplie les chemins que peut suivre le système dans son espace d'états.

Concernant le second point, considérons l'évolution du système à partir de l'état initial 011 avec mise à jour de tous les éléments à chaque itération (donc sans chaos). On peut penser a priori que l'évolution va mener au point fixe 000 et y rester. Mais cela n'est pas

le cas avec les retards car lorsqu'un élément est mis à jour, les valeurs des autres éléments qui sont utilisées peuvent être antérieures à leurs valeurs courantes. Ainsi, chaque élément évolue avec sa propre image globale du système. L'exemple de la Figure 3.1 montre que ces retards peuvent provoquer des évolutions inattendues car les éléments peuvent évoluer à partir d'images globales du système qui n'ont pas encore été atteintes réellement.

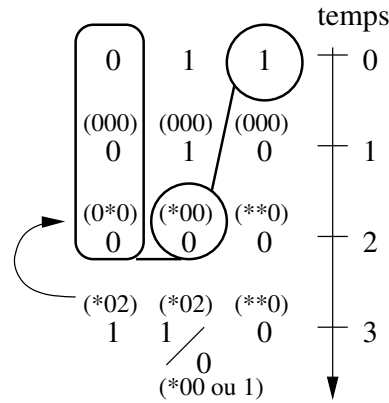


FIG. 3.1 – Évolution synchrone avec retards à partir de l'état 011 menant à un état (100 ou 110) non atteignable dans une évolution synchrone sans retards. Les chiffres entre parenthèses indiquent les retards utilisés (* pour n'importe lequel).

Par souci de clarté, le mode d'évolution synchrone a été considéré jusque là mais l'influence des retards se retrouve de la même façon dans le mode chaotique. Pour le vérifier, il suffit de reprendre l'état initial 011. En mode chaotique, les successeurs possibles de cet état sont 011 et 010. Comme les éléments ne peuvent rester indéfiniment sans être mis à jour, le troisième élément devra évoluer à un instant donné et l'état global passera alors en 010. De même, les successeurs possibles de 010 sont 000 et lui-même. Donc, en suivant le même raisonnement, l'état 000 finira par être atteint. Une fois dans cet état, le système y restera indéfiniment. Ainsi, à partir de l'état 011, le système converge vers le point fixe 000 en mode chaotique. Par contre, si les retards sont inclus, il suffit de reprendre la Figure 3.1, qui est un cas particulier d'évolution chaotique, pour voir qu'un état non atteignable en mode chaotique peut l'être en mode asynchrone. Cependant, il ne faut pas en conclure que l'influence des retards sur le mode chaotique ne vient que d'un cas particulier d'exécution, la Figure 3.2 montre que l'on peut obtenir le même comportement avec une évolution réellement chaotique.

On peut donc se rendre compte de la grande complexité d'évolution qui est induite par le mixage du chaos et des retards.

Pour analyser de manière plus formelle ces systèmes, certaines définitions et outils supplémentaires sont nécessaires. Il paraît donc utile d'effectuer un petit rappel des notions de base attachées à ce domaine. Pour rester clair et concis, les notions plus spécifiques aux travaux présentés dans la suite sont données dans les parties correspondantes.

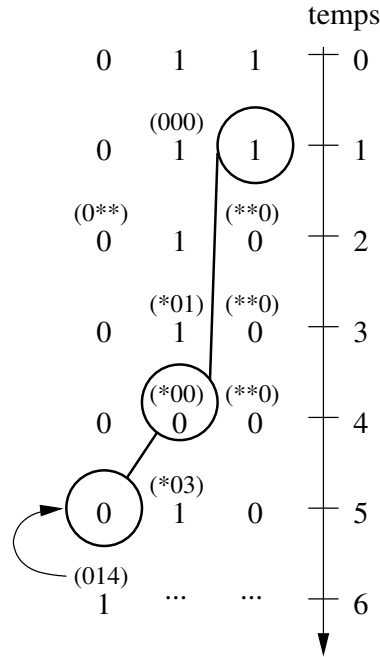


FIG. 3.2 – Évolution asynchrone à partir de l'état 011 menant à un état (100 ou 110) non atteignable dans une évolution chaotique. Les retards ne sont mentionnés que lors des mises à jour des éléments.

3.1.2 Définitions et outils

Une grande partie des éléments présentés ici sont issus des livres de F.Robert [124, 125] et ne sont pas spécifiques au mode d'évolution asynchrone.

Pour étudier la dynamique du système, il est nécessaire de pouvoir en comparer deux états et cela requiert une notion de distance, d'où la définition suivante.

Définition 3.1 *La distance vectorielle entre deux états du système $x = (x_1, \dots, x_n)$ et $y = (y_1, \dots, y_n)$ est définie par :*

$$d(x, y) = (\delta_1(x_1, y_1), \dots, \delta_n(x_n, y_n))$$

où

$$\delta_i(x_i, y_i) = \begin{cases} 1 & \text{si } x_i \neq y_i \\ 0 & \text{si } x_i = y_i. \end{cases}$$

De même, cette distance peut être étendue à la décomposition par blocs :

$$d(X, Y) = (d_1(X_1, Y_1), \dots, d_m(X_m, Y_m)),$$

où

$$d_i(X_i, Y_i) = \begin{cases} 1 & \text{si } X_i \neq Y_i \\ 0 & \text{si } X_i = Y_i. \end{cases}$$

Il est aussi important de faire la distinction entre deux graphes particuliers qui peuvent être associés à un système dynamique discret : le *graphe de connexion* et le *graphe d'itération*.

Le graphe de connexion indique les interdépendances entre les éléments du système. Ses sommets sont les éléments du système et ses arêtes indiquent les communications d'information entre ceux-ci. Ce graphe est orienté car les dépendances entre les éléments ne sont pas obligatoirement symétriques. Ainsi, une flèche allant d'un élément x_i vers un élément x_j indique que la fonction d'évolution f_j de x_j dépend de x_i . Ce graphe est donc implicitement défini par la donnée de la fonction d'évolution f du système. Une manière formelle de le représenter est la matrice de connexion.

Définition 3.2 *Si l'on considère un système dynamique discret à n éléments dont l'évolution est donnée par $f = (f_1, \dots, f_n)$ où f_i est la fonction d'évolution de l'élément i , alors la matrice booléenne $B(f)$ est définie pour chaque élément b_{ij} , $i, j \in \{1, \dots, n\}$ par :*

$$b_{ij} = \begin{cases} 1 & \text{si } f_i \text{ dépend de l'élément } j \\ 0 & \text{sinon} \end{cases}$$

La notion de contraction vue en 2.1.2 a alors une équivalence dans le cas discret.

Définition 3.3 *La fonction f d'évolution d'un système dynamique discret est une contraction si la matrice associée $B(f)$ a un rayon spectral nul.*

Il faut rappeler que les valeurs propres d'une matrice booléenne sont restreintes à zéro ou un et que le calcul de son rayon spectral est assez simple. Pour avoir un rayon spectral nul, la puissance $n^{\text{ème}}$ de la matrice doit être nulle, ce qui est équivalent à l'existence d'une matrice de permutation P telle que $P^t.B.P$ soit triangulaire inférieure stricte. Un rayon spectral nul correspond en fait à un graphe de connexion sans cycle. La notion de contraction prend alors tout son sens dans la proposition suivante.

Proposition 3.1 *Si la fonction f d'évolution d'un système dynamique discret est une contraction, alors f a un unique point fixe $x^* \in E$ et le système converge vers ce point fixe quelle que soit son initialisation et son mode d'évolution.*

Cependant, ce résultat a une portée relativement restreinte ici puisque la plupart des systèmes comportent des cycles dans leur graphe de connexion. Nous verrons dans la partie 3.4 comment contourner ce problème. De plus, il est parfois nécessaire que le système ait plusieurs points fixes, comme par exemple les réseaux neuronaux à mémoire associative.

En ce qui concerne le graphe d'itération, il décrit la dynamique complète du système. Ses sommets sont tous les états possibles du système et chaque arête indique une transition possible d'un état vers un autre. Là aussi, ce graphe est orienté. Contrairement à son homologue précédent, il n'est généralement pas possible de le construire entièrement, surtout

lorsque le nombre d'éléments du système, et donc d'états, est important. Cependant, on fait généralement une distinction entre les graphes d'itération en mode synchrone et asynchrone. En effet, les éléments n'étant pas obligatoirement mis à jour à chaque itération du mode asynchrone, on ne fait apparaître dans le graphe d'itération que celles qui sont représentatives de l'évolution réelle du système et qui sont définies relativement aux périodes de mise à jour.

Définition 3.4 *Soit un système discret à n éléments, les périodes de mise à jour sont définies par la séquence strictement croissante d'entiers $\{p_l\}_{l \in \mathbb{N}}$ définie comme suit :*

- $p_0 = 0$
- $p_{l, (l > 0)}$ est l'itération minimale à laquelle tous les éléments du système ont été mis à jour au moins l fois

Ainsi, une période de mise à jour correspond à un intervalle de temps minimal $[t, t + \Delta t]$, $t, \Delta t \in \mathbb{N}$ dans lequel tous les éléments du système sont mis à jour au moins une fois.

La notion de graphe d'itération permet aussi d'introduire les notions de *séquence itérative* et de *cycle itératif*.

Définition 3.5 *Soit f la fonction d'évolution d'un système dynamique discret, alors la suite d'états $U = (u^1, u^2, \dots, u^k)$ est une séquence itérative de longueur k de f si et seulement si :*

$$f(u^1) = u^2, \quad f(u^2) = u^3, \quad \dots, \quad f(u^{k-1}) = u^k$$

Définition 3.6 *Soit f la fonction d'évolution d'un système dynamique discret, alors la suite d'états $U = (u^1, u^2, \dots, u^k)$ est un cycle itératif de longueur k si et seulement si U est une séquence itérative et $f(u^k) = u^1$.*

Un cycle de longueur un correspond donc à un point fixe du système. Une autre définition utile est celle des *attracteurs* d'un système dynamique. On peut les définir comme un sous-ensemble d'états duquel le système ne sort plus une fois qu'il l'a atteint. Ils peuvent être réduits à un seul élément, ce qui nous ramène à la notion de point fixe. Lorsqu'ils comportent plus d'un élément, ils peuvent être cycliques ou non selon le mode d'évolution utilisé. Enfin, un même système peut comporter plusieurs attracteurs de types éventuellement différents.

3.2 État de l'art

Les systèmes dynamiques discrets représentent un outil de modélisation et d'analyse incontournable pour de nombreux systèmes, qu'ils soient naturels ou artificiels. Ces systèmes sont particulièrement adaptés à l'outil informatique puisque les machines sont encore largement représentées en suivant le modèle de la machine de Turing avec variables d'état et temps discrets [137].

On retrouve parmi les principaux contributeurs à ce domaine beaucoup de ceux qui ont travaillé sur le domaine continu [115, 59, 135, 69] puisque certains de leurs travaux ont une portée générale. D'ailleurs, il est assez rare de trouver des auteurs qui se focalisent sur un seul cadre d'étude des systèmes dynamiques. Mais certains auteurs ont spécifiquement travaillé sur les systèmes discrets comme F.Robert [123, 124, 125] qui a apporté une contribution importante à l'étude de la convergence des itérations discrètes chaotiques, cas particulier des itérations asynchrones. C'est à cette occasion qu'il a introduit la notion de distance vectorielle élément à élément. D'autres études ont suivi sur les problèmes de convergence dans des contextes plus généraux et en s'intéressant notamment à la description des attracteurs autour des points fixes [119, 57, 50].

Un des domaines où ces systèmes ont été très largement utilisés est celui des réseaux neuronaux. En effet, bien que le contexte d'étude le plus souvent utilisé dans ce domaine soit celui de systèmes hybrides avec un temps discret et des valeurs continues (voir par exemple [103, 70, 139]), bon nombre de ces réseaux peuvent être vus comme une sous-classe des systèmes dynamiques discrets, notamment les réseaux de neurones récurrents à seuils, tels que les réseaux de Hopfield [99, 100]. La plupart des études menées dans ce domaine portent sur la stabilisation de ces réseaux et/ou l'absence de cycles [92, 73, 114, 72, 130, 140] selon la dynamique utilisée. L'asynchronisme a aussi suscité l'intérêt des chercheurs de ce domaine et a fait l'objet d'études spécifiques telles que [70, 104] utilisant le modèle d'asynchronisme défini par M.Takeda et J.W.Goodman dans [134], dans lequel les délais ne sont pas pris en compte. Cependant, un modèle plus général de réseaux totalement asynchrones, similaire à celui présenté au Chapitre 1, a été caractérisé par A.V.M.Herz et C.M.Marcus dans [97]. Ce dernier s'est aussi intéressé à la dynamique globale des réseaux [113, 112] mais dans [97], lui et Herz arrivent à la conclusion que l'analyse de la stabilité des réseaux dans le cas le plus général d'asynchronisme reste un problème ouvert.

Ainsi, on peut voir qu'il reste encore beaucoup à faire sur l'étude des systèmes dynamiques asynchrones dans le cadre discret. Les résultats de convergence restent partiels dans le cas le plus général des systèmes totalement asynchrones, notamment lorsque ceux-ci possèdent plusieurs attracteurs. C'est donc sur ce thème que j'ai orienté mes travaux de recherche en étudiant le comportement local et global de ces systèmes puis en travaillant sur le mixage des modes synchrone et asynchrone de façon à stabiliser le comportement global du système tout en conservant un maximum d'asynchronisme.

3.3 Convergence basée sur des propriétés locales

Dans la lignée des travaux menés par F.Robert [124, 125] sur les attractions locales dans les voisinages compacts d'un point fixe et par D.Pellegrin [119] sur des algorithmes de vérification de certaines propriétés d'attraction, l'objectif de mes travaux [14, 10] a été d'identifier les initialisations d'un système discret totalement asynchrone qui mènent à une stabilisation de celui-ci sur un point fixe donné. Aucune condition particulière n'est formulée sur le système.

Il peut donc contenir plusieurs cycles et points fixes.

Le résultat théorique obtenu et l'algorithme de test de la convergence qui en a été déduit sont respectivement présentés dans cette partie avec un exemple d'application à un réseau booléen de Hopfield. Toutefois, avant de les présenter chacun en détails, il est nécessaire de définir certaines notions supplémentaires utilisées dans le résultat théorique.

3.3.1 Notions supplémentaires

Les notions nécessaires, en plus de celles déjà présentées précédemment, sont le voisinage d'un état et la dérivée discrète.

Définition 3.7 Soit $X = (X_1, \dots, X_m) \in E$, un état d'un système discret à n éléments décomposé en m blocs. On appelle voisinage de X tout sous-ensemble de E s'écrivant comme un produit cartésien et contenant X :

$$X \in V = \prod_{i=1}^m V_i \quad \text{avec} \quad V_i \subseteq E_i, \quad i = 1, \dots, m$$

Définition 3.8 Soit $X = (X_1, \dots, X_{j-1}, X_j, X_{j+1}, \dots, X_m) \in E$, un état d'un système discret à n éléments décomposé en m blocs. Alors \tilde{X}^j correspond à l'ensemble des états $(X_1, \dots, X_{j-1}, Y_j, X_{j+1}, \dots, X_m)$ tels que $Y_j \neq X_j$.

Définition 3.9 Soit un système discret à états dans E , décomposé en m blocs et dont l'évolution est définie par $F = (F_1, \dots, F_m)$. La dérivée discrète de F en $X \in E$ est une matrice carrée $F'(X)$ de dimension $m \times m$ dont les éléments sont définis par :

$$F'_{ij}(X) = \begin{cases} 1 & \text{si } \exists Y \in \tilde{X}^j \text{ tel que } F_i(X) \neq F_i(Y) \\ 0 & \text{si } \forall Y \in \tilde{X}^j \quad F_i(X) = F_i(Y) \end{cases}$$

La dérivée discrète en X est l'analogue de $\frac{\partial F}{\partial X}$ dans \mathbb{R}^m . Elle représente les variations de F autour de l'état X .

3.3.2 Résultat théorique

Un résultat classique basé sur la dérivée discrète a été donné dans le cadre des systèmes booléens (voir [125]) et permet de déduire une partie des initialisations d'un système dynamique discret totalement asynchrone qui vont mener à un point fixe donné.

Mes travaux [14] et [10] étendent ce résultat en permettant d'une part de trouver plus d'initialisations vérifiant cette contrainte et d'autre part de passer des systèmes booléens aux systèmes discrets quelconques. Cependant, pour rester clair et concis, seul le résultat le

plus général obtenu dans ces travaux est présenté ici. Ce résultat englobe aussi ceux obtenus dans [56, 57] sur les contractions et dans [50] sur les systèmes booléens.

Théorème 3.1 *Soit un système discret à n éléments décomposé en m blocs et dont l'évolution est définie par $F = (F_1, \dots, F_m)$ dans l'espace $E = \prod_{i=1}^m E_i$. Soit X^* un point fixe de F et $V = \prod_{i=1}^m V_i$ un voisinage de X^* . Si l'on a :*

(a) $F(V) \subset V$

(b) *la matrice $M = \sup_{z \in V} F'(z)$ où \sup est pris élément par élément, est une contraction. Alors, en notant i_1, \dots, i_q les indices des colonnes nulles de M (au moins une par (b)) et en définissant :*

$$W^k = V_1 \times \dots \times V_{i_k-1} \times E_{i_k} \times V_{i_k+1} \times \dots \times V_m$$

alors toute exécution totalement asynchrone de ce système atteint l'état X^ en au plus $p_1 + p_m$ itérations pour tout état initial X^0 tel que $X^0 \in \bigcup_{k=1}^q W^k$.*

La preuve de ce résultat est donnée dans [10]. Pour bien comprendre l'étendue de ce théorème, un exemple de son application est présenté sur un système discret à trois éléments à valeurs dans $\{0, 1, 2\}$ dont la fonction d'évolution est donnée dans la Table 3.2.

x	$F(x)$	x	$F(x)$	x	$F(x)$
000	000	100	021	200	012
001	000	101	021	201	012
002	000	102	021	202	210
010	001	110	022	210	211
011	002	111	021	211	012
012	202	112	222	212	212
020	001	120	020	220	012
021	001	121	020	221	012
022	001	122	020	222	112

TAB. 3.2 – Fonction F du système discret à valeurs dans $E = \{0, 1, 2\}^3$

Si l'on considère le voisinage $V = \{0, 1\} \times \{0, 2\} \times \{0, 1\}$ de l'état $(0, 0, 0)$, ce voisinage se réécrit explicitement :

$$V = \{(0, 0, 0), (0, 0, 1), (0, 2, 0), (0, 2, 1), (1, 0, 0), (1, 0, 1), (1, 2, 0), (1, 2, 1)\}$$

et il est nécessaire de calculer les dérivées discrètes en chaque état de V pour calculer M . Pour rester concis, on ne détaille que le calcul de $F'(0, 0, 0)$ avant de donner le résultat final. En fait, le calcul de la dérivée discrète peut être effectué colonne par colonne. En effet, la colonne j correspond en fait au \sup des distance entre $F(X)$ et les images par F des éléments de \tilde{X}^j . Ainsi, pour la colonne 0 de $F'(0, 0, 0)$ on calcule

$$d(F(0, 0, 0), F(1, 0, 0)) = d((0, 0, 0), (0, 2, 1)) = (0, 1, 1)$$

et

$$d(F(0, 0, 0), F(2, 0, 0)) = d((0, 0, 0), (0, 1, 2)) = (0, 1, 1)$$

pour obtenir

$$F'(0, 0, 0) = \begin{pmatrix} 0 & . & . \\ 1 & . & . \\ 1 & . & . \end{pmatrix}$$

En répétant le processus sur les autres colonnes on obtient

$$F'(0, 0, 0) = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Enfin, en répétant ces calculs sur les autres états de V , on arrive à

$$M = \sup_{z \in V} \{F'(z)\} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Seule la colonne 3 de M est nulle, on a donc $i_1 = i_q = 3$ et

$$W^1 = \{0, 1\} \times \{0, 2\} \times E_3 = \{0, 1\} \times \{0, 2\} \times \{0, 1, 2\}$$

et le Théorème 3.1 implique que toute exécution totalement asynchrone initialisée avec un élément de W^3 converge vers le point fixe $(0, 0, 0)$. Cela inclut les états $(0, 0, 2)$, $(0, 2, 2)$, $(1, 0, 2)$ et $(1, 2, 2)$ alors que leurs dérivées ne sont pas des contractions :

$$F'(0, 0, 2) = F'(0, 2, 2) = F'(1, 0, 2) = F'(1, 2, 2) = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

Ces états ne sont donc pas dans un voisinage contractant (tel que la matrice M associée soit contractante) de $(0, 0, 0)$. L'intérêt du Théorème 3.1 est donc de détecter aussi la convergence pour des états qui ne sont pas dans un voisinage contractant d'un point fixe mais seulement dans le voisinage premier de celui-ci (une seule différence avec un des éléments du voisinage), comme illustré dans la Figure 3.3. De tels états n'étaient pas détectés dans les résultats théoriques précédents.

3.3.3 Algorithme de test

À partir du Théorème 3.1, une procédure de test de la convergence, donnée dans l'Algorithme 19, a été mise au point. Elle n'a pas vocation à remplacer l'utilisation directe du système qu'elle teste mais à en étudier le comportement. Cependant, dans certains cas, il est plus rapide d'utiliser cette fonction que le système lui-même. D'autant que la convergence du système à partir d'un état donné n'est jamais complètement assurée par une série d'exécutions de celui-ci, quel qu'en soit le nombre.

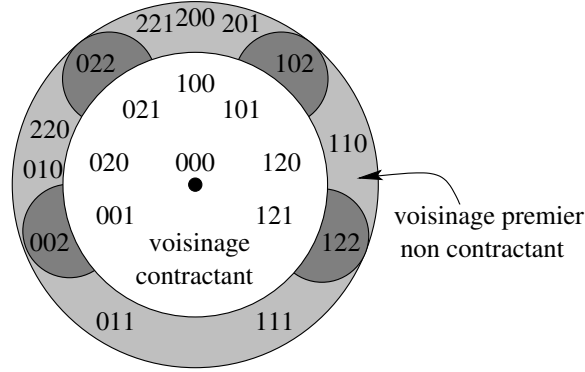


FIG. 3.3 – États supplémentaires (gris foncé) menant au point fixe $(0,0,0)$ autour de son voisinage contractant.

La fonction de test prend en paramètres deux états qui sont respectivement un état initial X^0 et un point fixe X^* du système et vérifie si les conditions du Théorème 3.1 sont remplies pour ces deux états. Si c'est le cas, cela signifie que le système initialisé avec X^0 converge vers X^* et la fonction retourne un booléen à **Vrai**. Dans le cas contraire, aucune hypothèse ne peut être formulée sur la convergence du système vers X^* à partir de X^0 et la fonction retourne **Faux**.

L'algorithme est divisé en deux grandes étapes. La première étape est basée sur les résultats antérieurs au Théorème 3.1 et effectue un test de convergence sans tenir compte des colonnes nulles de la matrice M . Son principe est de construire un voisinage V de X^* contenant X^0 et vérifiant les propriétés (a) et (b) du Théorème 3.1. Pour cela, on part du voisinage minimal de X^* contenant X^0 , construit à partir de leurs différences élément à élément, et on l'étend si nécessaire jusqu'à obtenir un voisinage qui contienne son image par F . Pour éviter les calculs inutiles, ce processus s'arrête avec un résultat **Faux** dès que la matrice M correspondante au voisinage courant n'est plus une contraction. Par contre, lorsque le processus aboutit à un voisinage contenant son image et ayant une matrice M contractante, alors la convergence est détectée.

La seconde étape n'est effectuée que si la première échoue et prend en compte les colonnes nulles de M . L'idée est de construire un voisinage plus petit que précédemment ne contenant pas X^0 mais dont l'extension d'une seule composante p à tout son domaine de valeur E_p permet d'inclure X^0 . Si un tel voisinage vérifie les conditions (a) et (b) du Théorème 3.1 et que la colonne p de la matrice M correspondante à ce voisinage est nulle alors il est assuré que X^0 fait converger le système vers X^* . Pour réaliser cela, on parcourt les indices p des éléments de X^0 qui sont différents de X^* et pour chacun on effectue un processus légèrement différent de celui vu à l'étape précédente. En effet, on ne part plus du voisinage minimal de X^* contenant X^0 mais d'un voisinage plus petit tenant compte de toutes leurs différences élément à élément sauf celle d'indice p . On obtient un voisinage de X^* qui ne contient pas X^0 uniquement à cause de la valeur de son élément d'indice p . Ensuite, ce voisinage est agrandi si nécessaire pour satisfaire à la condition (a) tout en veillant à ce que la matrice M' correspondante soit

Algorithme 19 Fonction TestConv($X^0 : \text{État}, X^* : \text{État}$) : booléen

résultat = Booléen indiquant si l'on peut conclure à la convergence ou pas, initialisé à Faux.

arrêt = Booléen indiquant l'arrêt du test, initialisé à Faux.

si $X^0 = X^*$ **alors**

résultat = Vrai

sinon

 si $\rho(F'(X^*)) = 0$ **alors**

 $V = \prod_{j=1}^m V_j$ où $V_j = \{X_j^0\} \cup \{X_j^*\}$

 répéter

 $M = \sup_{z \in V} F'(z)$

 si $\rho(M) = 0$ **alors**

 si $F(V) \subset V$ **alors**

résultat = Vrai

sinon

 $S = (\prod_{j=1}^m S_j) \subseteq E$ tel que $V \cup F(V) \subseteq S$ et $Card(S)$ est minimal

 $V = S$

 fin si

 sinon

arrêt = Vrai

fin si
jusqu'à arrêt = Vrai **ou** résultat = Vrai

si résultat = Faux **alors**

 répéter pour chaque p tel que $X_p^0 \neq X_p^*$

 $V' = \prod_{j=1}^m V'_j$ où $V'_j = \{X_j^0\} \cup \{X_j^*\}$ pour $j \neq p$ et $V'_p = \{X_p^*\}$

arrêt = Faux

répéter

 $M' = \sup_{z \in V'} F'(z)$

 si $\rho(M') = 0$ **et** $V^0 \notin V'$ **et** la colonne p de M' est nulle **alors**

 si $F(V') \subset V'$ **alors**

résultat = Vrai

sinon

 $S' = (\prod_{j=1}^m S'_j) \subseteq E$ tel que $V' \cup F(V') \subseteq S'$ et $Card(S')$ est minimal

 $V' = S'$

 fin si

 sinon

arrêt = Vrai

fin si

 jusqu'à arrêt = Vrai **ou** résultat = Vrai

 jusqu'à résultat = Vrai **ou** tous les p ont été parcourus

fin si
fin si
fin si

 retourner résultat

contractante. Il y a alors deux possibilités pendant ce processus d'agrandissement selon que X^0 se retrouve ou non dans V' . S'il se retrouve dedans, on revient alors au contexte de la première étape du test et il n'est donc pas nécessaire de continuer dans cette voie. Dans le cas contraire, seul l'élément p de X^0 l'exclu de V' et il faut vérifier que la colonne p de M' est nulle pour savoir si X^0 mène malgré tout à X^* . Enfin, si ce traitement est effectué sans succès pour chacun des éléments différents entre X^0 et X^* , alors on ne peut pas conclure sur la convergence à partir de X^0 .

On peut remarquer que la taille des ensembles V et V' augmente exponentiellement en fonction du nombre de différences entre l'état initial et le point fixe considérés. Cet algorithme peut donc nécessiter de grandes quantités de mémoire et de calculs. Une version parallèle peut donc être envisagée pour traiter des problèmes de taille importante.

3.3.4 Application à un réseau booléen de Hopfield

Les réseaux de Hopfield sont largement utilisés en reconnaissance de formes. Leur utilisation appliquée a suscité de nombreuses études sur leur dynamique dans divers contextes comme nous avons pu le voir dans la partie 3.2. Cependant, la plupart de ces travaux adoptent une approche globale ou utilisent des hypothèses spécifiques aux réseaux de Hopfield. Le Théorème 3.1 peut paraître plus restreint que certains de ces résultats mais il s'applique à une classe plus large de systèmes dynamiques et ne fait aucune hypothèse sur les propriétés du système. Dans le cadre des réseaux de Hopfield, cela implique notamment qu'ils peuvent avoir plusieurs attracteurs et qu'aucune contrainte n'est posée sur la matrice représentant les poids des neurones. L'application de la fonction de test présentée dans l'Algorithme 19 est réalisée sur un exemple de réseau de Hopfield.

On considère ici un réseau de Hopfield booléen à n neurones. Un tel réseau est spécifié par la fonction d'activation de ses neurones et le réseau d'interconnexion entre ceux-ci qui est décrit par une matrice carrée symétrique de dimension $n \times n$ à coefficients réels. Pour rester dans le contexte d'évolution purement lié à ces réseaux, on n'applique pas de décomposition par blocs, ce qui revient aussi à faire une décomposition en n blocs de un neurone chacun. On construit un réseau pouvant mémoriser k motifs $X^{*1}, X^{*2}, \dots, X^{*k} \in \{0, 1\}^n$ en utilisant la règle de Hebb. Celle-ci a été choisie car c'est la plus communément utilisée mais toute autre règle aurait pu être utilisée. La matrice d'interconnexion P est alors calculée de la façon suivante :

$$P = L.L^t \quad \text{où} \quad L = (X^{*1}, X^{*2}, \dots, X^{*k})$$

L est la matrice dont les colonnes sont les motifs à mémoriser.

Et la fonction d'activation des neurones est :

$$F(X) = H(P.X - S), \quad X = (X_1, \dots, X_n) \in \{0, 1\}^n$$

où S est le vecteur contenant les seuils des neurones et $H(X)$ est la version vectorielle de la fonction de Heaviside appliquée élément par élément au vecteur X . Il faut noter que la

construction d'un tel réseau n'assure pas que les motifs à mémoriser sont les seuls points fixes du système. La présence d'états parasites, mixages des motifs initiaux, n'est pas propre à la technique de construction utilisée mais est un problème général des réseaux de Hopfield. En pratique, il est souvent nécessaire de faire un compromis entre le nombre de motifs que l'on souhaite mémoriser et le nombre d'états parasites générés. Cela fait de ces réseaux un exemple encore plus intéressant sur lequel appliquer le test de convergence.

Une fois le réseau construit, on peut utiliser la fonction **TestConv()** décrite précédemment pour vérifier si ce réseau initialisé avec un état donné X^0 converge vers un des motifs $X^{*i}, i \in \{1, \dots, k\}$ en mode totalement asynchrone. Étant donné que cette fonction ne teste la convergence que vers un seul point fixe donné, l'Algorithme 20 est utilisé pour tester successivement la convergence vers chacun des motifs mémorisés à partir de l'initialisation spécifiée. De plus, pour accélérer le test, les motifs ayant le moins de différences avec X^0 sont testés en premier car ils nécessitent moins de calculs et ont une probabilité plus importante d'être le résultat de l'évolution du réseau à partir de X^0 .

Algorithme 20 Algorithme de test spécifique à un réseau de Hopfield

résultat = Booléen indiquant le résultat du test de convergence, initialisé à Faux.

X^0 = État initial à partir duquel la convergence est testée.

i = Numéro du motif vers lequel la convergence est testée, initialisé à 0.

Trier les motifs dans l'ordre croissant du nombre de différences avec X^0

répéter

$i = i + 1$

 résultat = TestConv(X^0, X^{*i})

jusqu'à $i = k$ **ou** résultat = Vrai

si résultat = Vrai **alors**

afficher "Convergence vers le motif ", i

sinon

afficher "Aucune conclusion possible"

fin si

Pour confirmer et évaluer l'étendue des résultats obtenus par cet algorithme, un programme de simulation d'un réseau de Hopfield avec dynamique totalement asynchrone a été développé. La fiabilité de ce simulateur a été confirmée par de multiples tests. Pour vérifier un résultat donné par l'Algorithme 20, on effectue une série de mille simulations avec la même initialisation de façon à obtenir un résultat statistique. Si deux simulations donnent des résultats différents alors on peut conclure que le réseau ne converge pas à partir de cette initialisation. De même on tient compte du nombre d'itérations effectuées lors d'une simulation pour vérifier si le nombre maximal d'itérations nécessaires à la convergence (précisé dans le Théorème 3.1) n'est pas dépassé. S'il est dépassé et qu'aucun point fixe n'a été atteint, cela signifie que le réseau est dans un cycle itératif.

La comparaison des résultats de l'algorithme de test et de la simulation a été effectuée sur un réseau à quinze neurones construit de façon à mémoriser les trois motifs donnés dans

la Table 3.3. Ensuite, plusieurs initialisations ont été testées et un échantillon représentatif des résultats obtenus est donné dans la Table 3.4. Aucune initialisation apparaissant dans ce tableau n'appartient à un voisinage contractant des trois motifs mémorisés.

X^{*1}	1	0	0	1	0	0	0	0	1	0	0	1	0	0	1
	○	●	●	○	●	●	●	●	○	●	●	○	●	●	○
X^{*2}	1	0	0	0	1	1	0	0	1	1	1	0	1	1	0
	○	●	●	●	○	○	●	●	○	○	○	●	○	○	●
X^{*3}	1	1	1	0	0	0	0	1	0	1	0	1	0	1	0
	○	○	○	●	●	●	●	○	●	○	●	○	●	○	●

TAB. 3.3 – Ensemble des motifs mémorisés (○ \equiv 1 et ● \equiv 0).

n°	X^0	nb de différences entre X^0 et			prédiction	simulation
		X^{*1}	X^{*2}	X^{*3}		
1	○○○●●●●●○○●●●●	2	9	6	X^{*1}	X^{*1}
2	●●●○○●●●○○●●●●	2	9	8	X^{*1}	X^{*1}
3	○●○○●○○●○○○●●●	9	2	9	X^{*2}	X^{*2}
4	○●●●●○○●○○●●○○●	6	3	8	X^{*2}	X^{*2}
5	○○○○●●●●○○○●○○●	6	9	2	X^{*3}	X^{*3}
6	●●●●●●●●●●●●●●	5	8	7	aucune	X^{*1}
7	○○○○○○○○○○○○○○○○	10	7	8	aucune	point fixe parasite
8	●●●○○●●○○●●●●●	5	6	9	aucune	divergence

TAB. 3.4 – Résultats de la fonction de test et de la simulation pour huit initialisations du réseau (○ \equiv 1 et ● \equiv 0).

Une première constatation est qu'il n'y a pas de contradictions entre les prédictions de l'Algorithme 20 et les simulations. Mais le résultat le plus intéressant est de pouvoir apprécier la portée du Théorème 3.1 et des algorithmes qui en ont été déduits. En effet, les résultats théoriques précédents n'auraient pas permis d'obtenir les prédictions des cinq premières lignes. Ces prédictions sont possibles grâce à la prise en compte des colonnes nulles de la matrice M . Par exemple, les résultats des deux premières initialisations peuvent être déduits de celui de l'état $Y=(○○○○●●●○○●●●●)$ appartenant au voisinage premier de X^{*1} (une seule différence avec X^{*1}) dont la matrice M associée est une contraction dans cet exemple. De plus, comme cette matrice M a toutes ses colonnes nulles, on en déduit que tous les vecteurs dans la voisinage premier de Y (une différence avec Y) mènent aussi à X^{*1} , même ceux qui ne sont pas dans un voisinage de X^{*1} ayant une matrice M contractante. C'est le cas des initialisations 1 et 2 dont respectivement le premier et le second élément diffère de Y . Le même principe a permis de trouver les prédictions des lignes 3, 4 et 5. Celles-ci ont été incluses dans la table pour indiquer que l'algorithme de prédiction n'est pas restreint à un seul motif mémorisé ou à un nombre de différences entre l'initialisation et le motif testé.

Enfin, les trois dernières initialisations montrent trois cas différents pour lesquels l'algorithme de test ne peut faire aucune prédiction.

Pour l'initialisation numéro 6, l'algorithme ne peut conclure bien que la convergence soit avérée vers X^{*1} . Cela vient du fait que les conditions du Théorème 3.1 sont suffisantes mais pas nécessaires pour avoir la convergence. En effet, ce résultat théorique prend en compte certains types de comportements du système mais pas tous. Dans le cas de l'état numéro 6, le réseau suit un comportement différent de ceux pris en compte dans ce résultat et la convergence ne peut donc pas être trouvée.

Dans les deux derniers cas, l'absence de prédiction est logique puisqu'il n'y a pas convergence vers un des motifs testés. Dans le cas numéro 7, il y a convergence mais vers un point fixe parasite. L'algorithme n'effectuant les tests qu'avec les motifs mémorisés qui sont les seuls points fixes connus au départ, il n'a donc pas pu trouver cette convergence. Cependant, même en ajoutant le test vers ce point fixe supplémentaire, aucune prédiction n'a pu être faite, ce qui n'est pas très surprenant vu le grand nombre de différences (cinq) entre l'initialisation et ce point fixe. En effet, pour que la convergence ait été détectée, il eut fallu que ce point fixe ait un voisinage contractant relativement grand (quatre différences), ce qui n'est généralement pas le cas pour les points fixes parasites. Enfin, pour le cas numéro 8, il n'y a pas convergence. Cela signifie que soit le réseau atteint un cycle soit qu'il n'arrive pas au même résultat d'une exécution à l'autre (points fixes différents et éventuellement cycles). Dans ce dernier cas, la prédiction reste correcte.

On voit donc que le Théorème 3.1 et l'algorithme de détection qui en a été déduit permettent d'étudier plus précisément la dynamique asynchrone de n'importe quel système dynamique discret donné.

3.4 Mixage synchronisme/asynchronisme

Si le travail précédent montre qu'il est possible de connaître assez précisément le comportement d'un système discret donné, la conception d'un système discret suivant un comportement donné en mode asynchrone reste très complexe. En effet, comme il a été décrit au début de ce chapitre, le problème lors de l'utilisation de l'asynchronisme vient de la part d'indéterminisme qu'il induit dans l'évolution du système. Cet indéterminisme implique généralement des comportements inattendus par rapport à la version synchrone du même système. Cette modification peut notamment rendre l'évolution du système instable en générant des divergences ou des cycles.

Dans ce contexte, l'asynchronisme pur peut être une voie sans issue et le recours à un mixage entre synchronisme et asynchronisme peut alors être un compromis très intéressant. Les recherches bibliographiques que j'ai effectuées sur ce sujet semblent indiquer que le mixage synchronisme/asynchronisme a été étudié dans d'autres domaines, notamment pour les protocoles de communication tolérants aux pannes dans les systèmes distribués temps réel (voir

par exemple [138]), mais apparemment pas dans le cadre des systèmes dynamiques discrets.

Pour débiter dans cette nouvelle voie, je me suis focalisé sur la stabilisation du comportement de systèmes relativement simples dont l'évolution en mode synchrone converge vers un unique point fixe quelle que soit leur initialisation. Malgré tout, des cycles itératifs peuvent apparaître dans l'évolution asynchrone de tels systèmes, empêchant leur convergence en un temps fini. Dans [16], l'objectif de mon travail a été de combiner synchronisme et asynchronisme dans de tels systèmes de façon à en assurer la convergence tout en conservant une part d'asynchronisme. La solution proposée est suivie d'une illustration de son application sur un exemple donné.

3.4.1 Stratégie de mixage

La solution utilisée repose sur trois résultats présentés précédemment. Le premier est la Proposition 3.1 qui stipule qu'un système converge vers son unique point fixe quelle que soit son initialisation et son mode de fonctionnement si sa matrice $B(f)$ associée est une contraction. Le second est le fait que la contraction de la matrice $B(f)$ correspond à un graphe de connexions sans cycle. Le dernier est le fait qu'un système décomposé par blocs est équivalent à un autre système non décomposé.

Il est important de noter qu'un système qui converge en mode synchrone vers un seul point fixe n'a pas obligatoirement sa matrice $B(f)$ contractante. La Proposition 3.1 donne une condition suffisante mais non nécessaire. C'est d'ailleurs une des raisons pour lesquelles un tel système peut ne pas conserver sa convergence en mode asynchrone. Cependant, comme il est possible, en utilisant la décomposition par blocs, de transformer un système discret en un autre, une façon d'assurer la convergence d'un système dont la matrice $B(f)$ n'est pas contractante est de le transformer en un autre système pour lequel la matrice $B(f)$ est contractante. Or, une telle matrice est obtenue lorsque le graphe de connexion du système est acyclique.

Ainsi, la solution qui est proposée pour la stabilisation du comportement d'un tel système en mode asynchrone consiste à décomposer celui-ci par blocs, de façon à obtenir un autre système avec un graphe de connexion acyclique entre les blocs. Or, pour ne plus avoir de cycles dans le graphe de connexion du nouveau système, il faut que chaque cycle de longueur maximale existant dans le système initial se retrouve à l'intérieur d'un même bloc. En effet, si tous les éléments formant un tel cycle sont dans le même bloc, alors ce cycle devient invisible au niveau des blocs. De plus, le fait que le cycle soit de longueur maximale est important. Prenons l'exemple de la Figure 3.4 qui exhibe une proposition de décomposition par blocs d'une partie d'un graphe de connexion, de la façon suivante : $X = ((x_1, x_2), (x_3), (x_4, x_5))$. On peut noter que le cycle $C_m = (x_1, x_2)$ est inclus dans un cycle plus grand $C_M = (x_1, x_2, x_3, x_4, x_5)$. Or, si l'on décompose le système comme indiqué sur cette figure, en plaçant dans un bloc uniquement les éléments de C_m , alors les autres éléments de C_M sont obligatoirement placés dans d'autres blocs (X_2 et X_3 ici) et le cycle formé par C_M reste visible entre ces blocs.

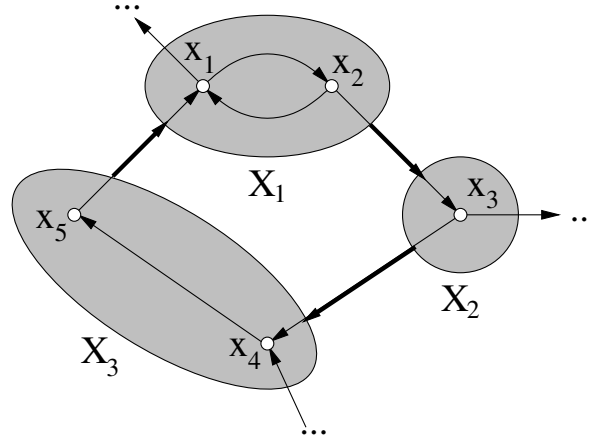


FIG. 3.4 – Décomposition par blocs conservant un cycle dans le graphe de connexion du nouveau système.

Enfin, comme nous l'avons vu au Chapitre 1, lors d'une décomposition par blocs d'un système, l'asynchronisme ne s'applique qu'entre les blocs et les éléments dans chaque bloc sont synchronisés entre eux. Pour conserver un maximum d'asynchronisme, la décomposition doit donc aussi être effectuée de façon à avoir le moins de synchronisations possibles entre les éléments, ce qui revient à avoir un maximum de blocs.

La stratégie finale est dictée par ces deux contraintes et consiste donc à :

- créer un bloc par cycle de longueur maximal
- créer un bloc pour chacun des éléments n'appartenant pas à un cycle

3.4.2 Application

Considérons un système booléen à six éléments ($x = (x_1, x_2, x_3, x_4, x_5, x_6) \in E = \{0, 1\}^6$) et le graphe de connexion donné dans la Figure 3.5.

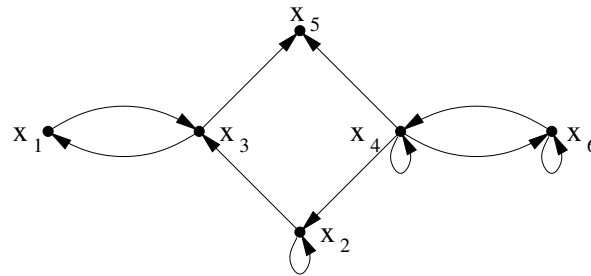


FIG. 3.5 – Graphe de connexion d'un système à six éléments.

Un tel graphe n'est pas spécifique à une seule fonction d'évolution mais correspond en fait à toute une classe de fonctions. Choisissons par exemple la fonction f définie de la façon

suivante :

$$f(x) = \begin{cases} f_1(x) = x_3 \\ f_2(x) = x_2 \overline{x_4} \\ f_3(x) = x_1 + \overline{x_2} \\ f_4(x) = x_4 \overline{x_6} + \overline{x_4} x_6 \\ f_5(x) = x_3 \overline{x_4} + \overline{x_3} x_4 \\ f_6(x) = \overline{x_4 + x_6} \end{cases}$$

On peut facilement vérifier que cette fonction correspond au graphe de la Figure 3.5. Le graphe d'itération correspondant à cette fonction est donné dans la Figure 3.6, dans laquelle chaque état $x = (x_1, x_2, x_3, x_4, x_5, x_6)$ est représenté par la valeur décimale du nombre booléen $x_1 x_2 x_3 x_4 x_5 x_6$ obtenu par la concaténation ordonnée des éléments. Ce graphe nous permet de constater que tous les états mènent au point fixe unique $44 \equiv (1, 0, 1, 1, 0, 0)$.

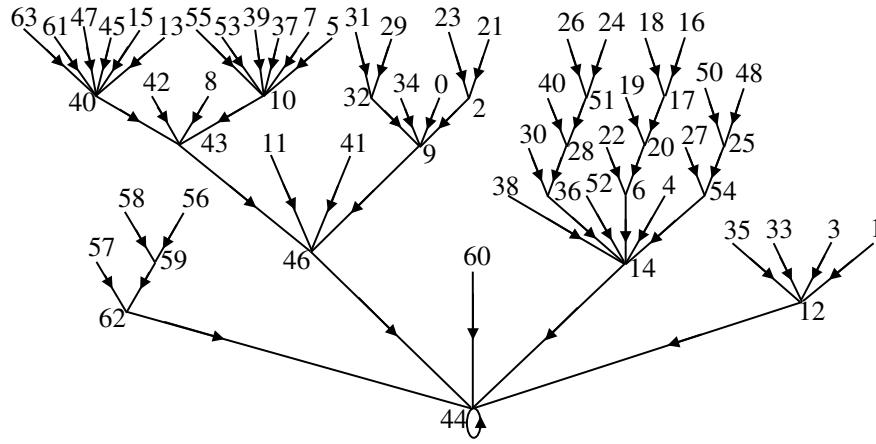
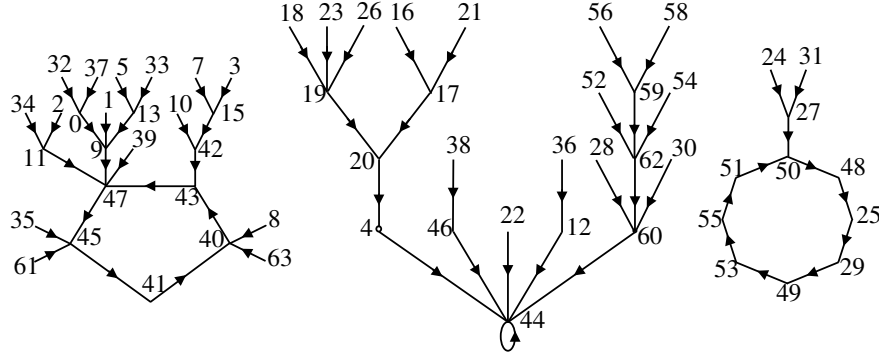


FIG. 3.6 – Graphe d'itération de la fonction f en mode synchrone.

Cependant, la matrice $B(f)$ associée a la forme suivante :

$$B(f) = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

et la présence de 1 sur la diagonale nous permet de déduire directement que son rayon spectral n'est pas nul. Ce n'est donc pas une contraction et l'on ne peut pas être assuré de la convergence de ce système en mode asynchrone. En fait, on peut voir dans la Figure 3.7, qui présente une partie du graphe d'itération asynchrone de ce système, que celui-ci comporte des cycles itératifs. Pour ne pas surcharger la figure, certaines transitions du graphe d'itération complet n'apparaissent pas.

FIG. 3.7 – Graphe d'itération partiel de la fonction f en mode asynchrone.

Pour conserver la stabilité de ce système en mode asynchrone, la stratégie de décomposition par blocs basée sur les cycles du graphe de connexion est utilisée et aboutit au système suivant :

$$X = \begin{pmatrix} X_1 = (x_1, x_3) \in \{0, 1\}^2 \\ X_2 = (x_4, x_6) \in \{0, 1\}^2 \\ X_3 = (x_2) \in \{0, 1\} \\ X_4 = (x_5) \in \{0, 1\} \end{pmatrix} \quad \text{et} \quad F(X) = \begin{pmatrix} F_1(X) = (f_1(X), f_3(X)) \\ F_2(X) = (f_4(X), f_6(X)) \\ F_3(X) = (f_2(X)) \\ F_4(X) = (f_5(X)) \end{pmatrix}$$

Le graphe de connexion associé à ce système, détaillé dans la Figure 3.8, est acyclique.

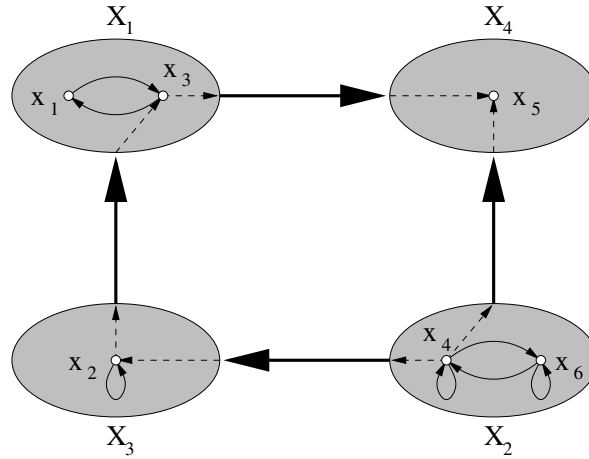


FIG. 3.8 – Graphe de connexion du système décomposé par blocs.

Il en résulte que la matrice $B(F)$ décrite ci-dessous peut être mise sous la forme d'une matrice triangulaire inférieure stricte par une suite de permutations ligne/colonne et a donc un rayon spectral nul.

$$B(F) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Cela assure que ce système a un point fixe unique et qu'il va converger vers ce point fixe quelle que soit son initialisation. Comme les F_i dépendent directement des f_i , un point fixe de f est aussi point fixe de F . Et comme il n'y en a qu'un seul, le système obtenu converge donc vers l'état $((1, 1), (1, 0), (0), (0))$ qui est la version décomposée par blocs du point fixe $(1, 0, 1, 1, 0, 0)$ de f .

Ainsi, en ne synchronisant que certains éléments entre eux, choisis de manière pertinente, on arrive à concevoir un système discret asynchrone qui a le même comportement global qu'un système discret synchrone donné. Cependant, le graphe de connexion d'un système donne une information assez grossière sur celui-ci et son utilisation ne permet pas toujours d'obtenir un mixage optimal ou même simplement un mixage. Il est clair que l'intérêt de cette étude réside plus sur la mise en évidence de la faisabilité de la stabilisation des systèmes dynamiques par mixage des modes synchrone et asynchrone que sur la précision de la solution proposée. Pour arriver à un nombre minimal de synchronisations, une étude plus précise prenant en compte les fonctions d'évolution est nécessaire.

3.5 Conclusion et perspectives

Comme il a été mentionné au début de ce chapitre, les systèmes discrets représentent non seulement un outil de modélisation très précis et puissant mais sont aussi, dans une certaine mesure, les systèmes de calcul utilisés dans les machines actuelles. Leur analyse et leur conception est donc primordiale. C'est donc dans ces deux voies que j'ai orienté mes recherches dans le cadre discret. Comme il a été montré tout au long de ce chapitre, celles-ci portent d'une part sur l'analyse du comportement des systèmes discrets asynchrones et d'autre part sur la conception de nouveaux systèmes combinant synchronisme et asynchronisme de façon à répondre à des exigences comportementales particulières.

La partie d'analyse a permis de mettre en évidence la complexité des comportements générés par l'asynchronisme total et d'en déduire des résultats théorique et pratique sur la détermination de la convergence d'un système pour un point fixe et une initialisation donnés. Cependant, même si ce résultat est plus précis que les précédents, il ne permet pas de décrire de manière exhaustive tous les cas de convergence. L'objectif sur ce thème serait donc d'arriver à une description complète de la dynamique de ces systèmes.

En ce qui concerne les systèmes hybrides synchrone/asynchrone, leur intérêt semble essentiel pour obtenir des systèmes efficaces tout en conservant un comportement stable. Les premiers résultats sur ce sujet sont prometteurs même si la solution proposée n'assure pas toujours un nombre minimal de synchronisations. Une voie de recherche très intéressante reste donc à explorer sur ce thème.

Enfin, ces travaux dans le cadre discret peuvent avoir des prolongements dans d'autres domaines assez variés, allant des réseaux de neurones, comme on l'a vu dans ce chapitre, aux systèmes de cryptographie.

Conclusion

Les systèmes dynamiques représentent un point clé de l'informatique parallèle. Ils peuvent se décliner sous la forme d'algorithmes itératifs permettant de résoudre une grande partie des problèmes rencontrés en calcul scientifique. Leur utilisation synchrone est très efficace sur les systèmes parallèles homogènes et localisés. Cependant, les évolutions récentes des réseaux de communication tendent à élargir les systèmes de calcul aux méta-grappes, ensembles de machines hétérogènes réparties sur plusieurs sites géographiques dans le monde. Ces systèmes ne sont pas adaptés au synchronisme, du fait des différences probables entre les vitesses des processeurs et les débits des liens de communication. Ainsi, pour répondre à cette évolution matérielle, il est nécessaire de faire évoluer la composante logicielle pour obtenir une exploitation optimale de ces nouveaux systèmes de calcul.

Ayant travaillé sur le parallélisme dès mes premières activités de recherche, j'ai naturellement orienté mes recherches sur cette évolution logicielle. En travaillant avec le Pr. Jacques Bahi, j'ai pu me rendre compte que l'asynchronisme présentait toutes les qualités requises pour effectuer cette évolution de manière quasiment naturelle et, qui plus est, très efficace, tout en proposant deux aspects d'étude complémentaires.

Le premier est le cadre continu. Ce cadre d'utilisation a une connotation relativement pratique puisqu'il aboutit directement aux algorithmes itératifs asynchrones qui sont utilisés pour la résolution de différents types de problèmes scientifiques, linéaires ou non. L'objectif de mes travaux sur ce thème a consisté à mettre en évidence l'intérêt que présente l'asynchronisme dans la programmation des méta-grappes de calcul mais aussi à proposer des solutions pour résoudre les problèmes apportés par l'utilisation de l'asynchronisme. Ainsi, en plus des schémas algorithmiques généraux, des mécanismes d'équilibrage de charge, de détection de la convergence et d'arrêt du processus global ont été proposés. De même, les aspects purement pratiques tels que les environnements de développement adaptés et les caractéristiques qui leurs sont nécessaires pour la mise en œuvre efficace des algorithmes asynchrones n'ont pas été négligés. Bien entendu, il reste plusieurs voies de recherche dans ce domaine, notamment en ce qui concerne les méthodes de multi-décomposition ou encore les solveurs linéaires. Mon objectif pour ce thème de recherche consiste donc à continuer mes travaux sur ces sujets.

Le second cadre est le discret. Étonnamment, bien que les machines actuelles soient dans une certaine mesure des systèmes discrets, ce cadre est surtout étudié de manière théorique. Il permet une modélisation précise de systèmes complexes, naturels ou non, mais aussi la

résolution de problèmes de nature discrète, tels que la reconnaissance de formes, particulièrement courants en informatique. La nature discrète permet une étude plus aisée des comportements complexes induits par l'asynchronisme. Ainsi, mes travaux dans ce domaine se sont portés sur la description de la dynamique des systèmes discrets asynchrones. L'expérience acquise lors de ces travaux m'a également permis d'aborder la conception de systèmes dynamiques répondant à un comportement attendu. Dans ce contexte, il a été montré que les systèmes hybrides, combinant synchronisme et asynchronisme, représentent une alternative très prometteuse, voire peut-être même incontournable. Cependant, à l'instar du cadre continu, il reste plusieurs pistes à explorer telles que la description complète de la dynamique d'un système asynchrone ou un mixage plus fin entre synchronisme et asynchronisme permettant de minimiser les synchronisations. De plus, d'autres thèmes de recherche, apparemment indépendants de celui-ci, peuvent nécessiter le recours aux systèmes dynamiques discrets totalement asynchrones. Un de ces thèmes, particulièrement intéressant, est celui de la cryptographie par chaos fondée sur des comportements physiques chaotiques qui ont un équivalent dans les systèmes discrets. On peut donc aisément imaginer l'intérêt scientifique d'études portant sur la combinaison de ces domaines. Ce type de travaux représente une suite logique à donner à l'ensemble de mes recherches.

Finalement, on peut constater que l'asynchronisme n'est pas seulement utile dans un cadre purement distribué mais peut également être utile dans un cadre d'utilisation séquentielle, par le biais de simulations. L'objectif de mes recherches à venir sera donc non seulement de poursuivre mes travaux sur l'amélioration des techniques d'analyse et de conception des systèmes asynchrones mais aussi de contribuer au développement des différentes formes de leur utilisation.

Liste des publications

Dépôts de logiciels

- [1] Sylvain Contassot-Vivier and Serge Miguet. Volter. *Agence pour la Protection des Programmes*. No : *IDDN.FR.001.460013.00.S.P.1996.000.21000*, 1996.
- [2] Jacques Bahi, Sylvain Contassot-Vivier, Libor Makovicka, Eric Martin, and Marc Sauget. Neurad. *Agence pour la Protection des Programmes*. No : *IDDN.FR.001.130035.000.S.P.2006.000.10000*, 2006.

Revue d'audience internationale

- [3] Jacques M. Bahi and Sylvain Contassot-Vivier. Basins of attraction in fully asynchronous discrete-time discrete-state dynamic networks. *IEEE Transactions on Neural Networks*, 17(2) :397–408, 2006.
- [4] J.M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *Journal of Supercomputing. Special Issue on Performance Modelling and Evaluation of Parallel and Distributed Systems*, 35(3) :227–244, 2006.
- [5] J. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5) :439–461, 2005.
- [6] R. Mathieu, E. Martin, R. Gschwind, L. Makovicka, S. Contassot-Vivier, and J. Bahi. Calculations of dose distributions using a neural network model. *Physics in Medicine and Biology*, 50(5) :1019–1028, 2005.
- [7] J. Bahi, S. Contassot-Vivier, and R. Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(4) :289–299, 2005.
- [8] J. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(1) :4–13, 2005.

- [9] Sylvain Contassot-Vivier, Frédéric Lombard, Jean-Marc Nicod, and Laurent Philippe. Evaluation of the diet hierarchical metacomputing architecture. *Parallel and Distributed Computing Practices : Special Issue on Parallel Numeric Algorithms on Faster Computers*, 5(4), December 2002.
- [10] Jacques M. Bahi and Sylvain Contassot-Vivier. Stability of fully asynchronous discrete-time discrete-state dynamic networks. *IEEE Transactions on Neural Networks*, 13(6) :1353–1363, 2002.
- [11] Eddy Caron, Serge Chaumette, Sylvain Contassot-Vivier, Frédéric Desprez, Eric Fleury, Claude Gomez, Maurice Goursat, Emmanuel Jeannot, Dominique Lazure, Frédéric Lombard, Jean-Marc Nicod, Laurent Philippe, Martin Quinson, Pierre Ramet, Jean Roman, Franck Rubi, Serge Steer, Frédéric Suter, and Gil Utard. Scilab to Scilab//, the OURAGAN Project. *Parallel Computing*, 11(27) :1497–1519, 2001.
- [12] Sylvain Contassot-Vivier and Serge Miguet. A Load-Balanced Algorithm For Parallel Digital Image Warping. *International Journal of Pattern Recognition and Artificial Intelligence*, 13(4) :445–463, 1999.
- [13] Sylvain Contassot-Vivier. A Load Balanced Parallel Ground Visualization Tool. *International Journal of Pattern Recognition and Artificial Intelligence*, 11(7) :1113–1127, 1997.

Revue d'audience nationale

- [14] Sylvain Contassot-Vivier and Jacques M. Bahi. Convergence dans les systèmes booléens asynchrones et application aux réseaux de hopfield. *Calculateurs Parallèles*, 13(1) :107–124, 2001.

Conférences internationales

- [15] J. Bahi, S. Contassot-Vivier, L. Makovicka, E. Martin, and M. Sauget. Neural network based algorithm for radiation dose evaluation in heterogeneous environments. In *Artificial Neural Networks - ICANN 2006*, volume 4132/2006 of *Lecture Notes in Computer Science*, pages 777–787, Athens, Greece, September 2006. Springer Berlin / Heidelberg.
- [16] A. Abbas, J. Bahi, S. Contassot-Vivier, and M. Salomon. Mixing synchronism / asynchronism in discrete-state discrete-time dynamic networks. In *4th International Conference on Engineering Applications and Computational Algorithms, DCDIS 2005.*, pages 524–529, Guelph, Canada, July 2005.
- [17] J. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing aiac algorithms. In *18th IEEE and ACM Int. Conf. on Parallel and Distributed Processing Symposium, IPDPS 2004*, pages 247b, 8 pages, Santa Fe, USA, April 2004. IEEE computer society press.

- [18] J. Bahi, S. Contassot-Vivier, and R. Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *17th IEEE and ACM int. conf. on International Parallel and Distributed Processing Symposium, IPDPS 2003*, pages 40a, 9 pages, Nice, France, April 2003. IEEE computer society press.
- [19] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Asynchronism for iterative algorithms in a global computing environment. In *The 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'2002)*, pages 90–97, Moncton, Canada, June 2002.
- [20] Sylvain Contassot-Vivier and Jean-Paul Rasson. Vectorization, matching and simplification of image contours in uncalibrated aerial stereo-vision. In *The EOS/SPIE Symposium on Remote Sensing : Image and Signal Processing for Remote Sensing VI*, pages 41–48, Barcelone, Spain, September 2000.
- [21] Sylvain Contassot-Vivier, Frédéric Lombard, Jean-Marc Nicod, and Laurent Philippe. Specification of a scilab meta-computing extension. In *ICPP Workshops : International Workshop on Metacomputing Systems and Applications*, pages 207–214, Toronto, Canada, August 2000.
- [22] Sylvain Contassot-Vivier and Serge Miguet. Optimization and Parallelization of a Geographical Stereo Vision Code. In *The Sixth International Conference in Central Europe on Computer Graphics and Visualization'98*, pages 65–72, University of West Bohemia, Campus Bory, Plzen, Czech Republic, February 1998.
- [23] Sylvain Contassot-Vivier and Serge Miguet. Parallel Geometric Transformations of Images. In *Fifth International Workshop on Parallel Image Analysis*, pages 13–30, Hiroshima, Japan, September 1997.
- [24] Sylvain Contassot-Vivier and Serge Miguet. Parallel Visualization of Texture-Mapped Digital Elevation Models. In *Fourth International Workshop on Parallel Image Analysis*, pages 269–282, Lyon, France, December 1995.

Conférences nationales

- [25] Marc Sauget, Sylvain Contassot-Vivier, Jacques Bahi, Eric Martin, and Libor Makovicka. Utilisation des réseaux de neurones pour la conception d'un code de calcul pour la dosimétrie en radiothérapie externe. In *22ièmes Journées des Laboratoires Associés de Radiophysique et de Dosimétrie*, Montbéliard, France, October 2005.
- [26] Marc Sauget, Eric Martin, Régine Gschwind, Libor Makovicka, Sylvain Contassot-Vivier, and Jacques Bahi. Développement d'un code de calcul dosimétrique basé sur les réseaux artificiels de neurones. In *44ièmes Journées Scientifiques de la Société Française de Physique Médicale*, Avignon, France, June 2005.
- [27] J. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. Asynchronisme et équilibrage de charge dans la grille de calcul. In *École d'hiver GRID 2002*, pages 365–373, Aussois, France, December 2002.

- [28] Sylvain Contassot-Vivier, Frédéric Lombard, Jean-Marc Nicod, and Laurent Philippe. Spécification de services sous CORBA pour une extension métacomputing de Scilab. In *RenPar'12*, pages 33–38, Besançon, France, June 2000.
- [29] Sylvain Contassot-Vivier. Calculs Parallèles pour la Visualisation de Terrains avec Textures. In *RenPar'8, Edition Spéciale, GDR-PRC Parallélisme, Réseaux et Systèmes*, pages 93–96, Bordeaux, France, May 1996.

Communications

- [30] Marc Sauget, Sylvain Contassot-Vivier, Jacques Bahi, Eric Martin, and Libor Makovicka. Évaluation de doses d'irradiation en milieux hétérogènes. In *Journées de L'ISIFC*, Besançon, France, October 2005.
- [31] Roland Mathieu, Éric Martin, Régine Gschwind, Libor Makovicka, Sylvain Contassot-Vivier, and Jacques Bahi. Utilisation des réseaux d'apprentissage neuronaux en radiothérapie externe. In *43èmes Journées scientifiques de la SFPM*, Montpellier, France, June 2004. prix de la meilleur communication affichée.
- [32] Roland Mathieu, Régine Gschwind, Éric Martin, Libor Makovicka, Sylvain Contassot-Vivier, and Jacques Bahi. Use of the neural networks in external radiotherapy. In *Current Topics in Monte Carlo Treatment Planning; Advanced Workshop*, Montréal, Canada, May 2004.
- [33] Roland Mathieu, Sylvain Contassot-Vivier, Libor Makovicka, Régine Gschwind, Éric Martin, and Jacques Bahi. Utilisation des rna en dosimétrie. In *20èmes Journées des LARD*, Clermont-Ferrand, France, October 2003.
- [34] Roland Mathieu, Sylvain Contassot-Vivier, Camille Guillerminet, Régine Gschwind, Libor Makovicka, and Jacques Bahi. Prospective de la planification des traitements radiothérapeutiques basée sur les réseaux de neurones. In *Journées Codes de calcul en radioprotection, radiophysique et dosimétrie*, Sochaux, France, October 2003. SFRP-SFPM-FIRAM.
- [35] Sylvain Contassot-Vivier. Parallel visualization of texture-mapped digital elevation models. In *Séminaire dans le cadre du projet PARALIN*, Département d'Ingénierie Mathématique, Université de Santiago, Chili, November 1996.

Rapports de recherche

- [36] Jacques Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Interest of the asynchronism in parallel iterative algorithms on meta-clusters. Research report, AND Team, LIFC, IUT de Belfort-Montbéliard, Belfort, France, 2005.

- [37] Jacques Bahi and Sylvain Contassot-Vivier. A convergence result on fully-asynchronous discrete-time discrete-state dynamic networks. Research report, AND Team, LIFC, IUT de Belfort-Montbéliard, Belfort, France, 2002.
- [38] E.Caron, P.Combes, S.Contassot-Vivier, F.Desprez, F.Lombard, J.-M.Nicod, M.Quinson, and F.Suter. A scalable approach to network enabled servers. Research Report 2002-21, Laboratoire de l'Informatique du Parallélisme (LIP), École Normale Supérieure de Lyon, France, May 2002.
- [39] E.Caron, S.Chaumette, S.Contassot-Vivier, F.Desprez, E.Fleury, C.Gomez, M.Goursat, E.Jeannot, D.Lazure, F.Lombard, J.M.Nicod, L.Philippe, M.Quinson, P.Ramet, J.Roman, F.Rubi, S.Steer, F.Suter, and G.Utard. Scilab to scilab//, the ouragan project. Research Report 2001-24, Laboratoire de l'Informatique du Parallélisme (LIP), École Normale Supérieure de Lyon, France, June 2001.
- [40] Sylvain Contassot-Vivier, Giosué Lo Bosco, and Chanh Dao Nguyen. Multiresolution approach for image processing. Technical report, Erasmus ICP-A-2007, Leiden, Pays-Bas, April 1996.
- [41] Sylvain Contassot-Vivier and Serge Miguet. Parallel visualization of textured-mapped digital elevation models. Technical Report RR1996-02, Laboratoire de l'Informatique du Parallélisme (LIP), École Normale Supérieure de Lyon, France, 1996.
- [42] T.Ludwig, R.Artiges, F.Avitabile, P.Baraduc, S.Contassot-Vivier O.Bournez, J.Cohen, E.Frejafon, E.Jeannot, G.Mounié N.D.Morelon, C.Perez, A.Pobla, C.Randriamaro, P.Rebreyend, M.Weber L-P. Tock, and J.V.Weizsäcker. The solitaire project – finding solutions for the game in parallel. Technical Report TR1994-01, Laboratoire de l'Informatique du Parallélisme (LIP), École Normale Supérieure de Lyon, France, July 1994.

Mémoires de stages

- [43] Sylvain Contassot-Vivier. *Calculs parallèles pour le traitement des images satellites*. PhD thesis, École Normale Supérieure, Lyon, France, February 1998.
- [44] Sylvain Contassot-Vivier. Visualisation parallèle de MNT texturés. Master's thesis, École Normale Supérieure, Lyon, France, June 1995.
- [45] Sylvain Contassot-Vivier. Détection de collisions sur machine parallèle. Rapport de stage de maîtrise, Université Laval, Québec, Canada, August 1994.
- [46] Sylvain Contassot-Vivier. Reconstruction 3D de terrains texturés. Rapport de stage de licence, École Normale Supérieure de Lyon, France, August 1993.

Bibliographie

- [47] Gheorghe Antonoiu and Pradip K. Srimani. A self-stabilizing leader election algorithm for tree graphs. *Journal of Parallel and Distributed Computing*, 34(2) :227–232, 1 May 1996.
- [48] Ken Arnold, James Gosling, and David Holmes. *The Java(TM) Programming Language*. Addison-Wesley, fourth edition, 2005.
- [49] Olivier Aumage, Guillaume Mercier, and Raymond Namyst. MPICH/Madeleine : a True Multi-Protocol MPI for High-Performance Networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, April 2001. IEEE.
- [50] J. Bahi. Boolean totally asynchronous iterations. *International Journal of Mathematical Algorithms*, 1 :331–346, 2000.
- [51] J. Bahi, S. Domas, and K. Mazouzi. Jace : a java environment for distributed asynchronous iterative computations. In *12-th Euromicro Conference on Parallel, Distributed and Network based Processing, PDP'04*, pages 350–357, Coruna, Spain, February 2004. IEEE computer society press.
- [52] J. M. Bahi, E. Griepentrog, and J. C. Miellou. Parallel treatment of a class of differential-algebraic systems. *SIAM Journal on Numerical Analysis*, 33(5) :1969–1980, October 1996.
- [53] J. M. Bahi, J.-C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3 and 4) :315–345, 1997.
- [54] J. M. Bahi, K. Rhofir, and J.-C. Miellou. Parallel solution of linear DAEs by multisplitting waveform relaxation methods. *Linear Algebra and Its Applications*, 3(332–334) :181–196, 2001.
- [55] Jacques M. Bahi. Asynchronous iterative algorithms for nonexpansive linear systems. *Journal of Parallel and Distributed Computing*, 60(1) :92–112, January 2000.
- [56] J.M. Bahi and C.J. Michel. Simulations of asynchronous evolution of discrete systems. *Simulation Practice and Theory*, 7 :309–324, 1999.
- [57] J.M. Bahi and C.J. Michel. Convergence of discrete asynchronous iterations. *International J. Computer Math.*, 74 :113–125, 2000.

- [58] D. H. Ballard, P. C. Gardner, and M. A. Srinivas. Graph problems and connectionist architectures. Technical Report 167, University of Rochester, Rochester, N.Y., March 1987.
- [59] G.M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25 :226–244, 1978.
- [60] Didier El Baz. A method of terminating asynchronous iterative algorithms on message passing systems. *Parallel Algorithms and Algorithms*, 9 :153–158, 1996.
- [61] Didier El Baz. *Contribution à l’algorithmique parallèle, le concept d’asynchronisme : étude théorique, mise en oeuvre, et application. HDR.* LAAS CNRS, Institut National Polytechnique de Toulouse, 1998.
- [62] Didier El Baz, Pierre Spiteri, Jean Claude Miellou, and Didier Gazen. Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. *Journal of Parallel and Distributed Computing*, 38(1) :1–15, 10 October 1996.
- [63] Bassem F. Beidas and George P. Papavassilopoulos. Distributed asynchronous algorithms with stochastic delays for constrained optimization problems with conditions of time drift. *Parallel Computing*, 21(9) :1431–1450, September 1995.
- [64] P. Beraldi and F. Guerriero. Parallel asynchronous implementation of the ϵ -relaxation method for the linear minimum cost flow problem. *Parallel Computing*, 23(8) :1021–1044, July 1997.
- [65] D. P. Bertsekas. Distributed asynchronous computation of fixed points. *Math. Programming*, 27 :107–120, 1983.
- [66] Dimitri P. Bertsekas and John N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *Conference Proceedings, 1989 International Conference on Supercomputing*, pages 461–470, Crete, Greece, June 5–9, 1989. ACM SIGARCH.
- [67] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation : Numerical Methods.* Prentice Hall, Englewood Cliffs NJ, 1989.
- [68] Dimitri P. Bertsekas and John N. Tsitsiklis. Parallel and distributed iterative algorithms : a selective survey. *Automatica*, 25 :3–21, 1991.
- [69] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice Hall, Englewood Cliffs, New Jersey, 1999.
- [70] A. Bhaya, E. Kaszkurewicz, and V.S. Kozyakin. Existence and stability of a unique equilibrium in continuous-valued discrete-time asynchronous hopfield neural networks. *IEEE Trans. Neural Networks*, 7(3) :620–628, 1996.
- [71] Kostas Blathras, Daniel B. Szyld, and Yuan Shi. Timing models and local stopping criteria for asynchronous iterative algorithms. *Journal of Parallel and Distributed Computing*, 58(3) :446–465, September 1999.
- [72] J. Bruck. On the convergence properties of the hopfield model. *Proc. IEEE*, 78(10) :1579–1585, 1990.

- [73] J. Bruck and J.W. Goodman. A generalized convergence theorem for neural networks. *IEEE Trans. Inform. Theory*, 34 :1089–1092, 1998.
- [74] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press Inc., New York, 1995.
- [75] Andréa S. Charão. *Multiprogrammation parallèle générique des méthodes de décomposition de domaine*. PhD thesis, Institut National Polytechnique de Grenoble, 2001.
- [76] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Its Applications*, 2 :199–222, 1969.
- [77] Victor Conrad and Yehuda Wallach. Iterative solution of linear equations on a parallel processor system. *IEEE Transactions on Computers*, C-26(9) :838–847, September 1977.
- [78] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2) :279–301, October 1989.
- [79] E. W. Dijkstra, W. H. J. Feijen, and A. J. M vanGasteren. Derivation of a termination detection algorithm for distributed computation. *Information Processing Letters*, 16(5) :217–219, 1983.
- [80] Mohamed El-Ruby, James Kenevan, Robert Carison, and Khalid Khalil. Leader election in distributed computing systems. In Naveed A. Sherwani, Elise de Doncker, and John A. Kapenga, editors, *Proceedings of Computing in the 90's*, volume 507 of *LNCs*, pages 350–356, Berlin, Germany, October 1991. Springer.
- [81] Robert Elsasser, Burkhard Monien, and Robert Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35 :305–320, 2002.
- [82] C. C. Foster. *Parallel Execution of Iterative Algorithms*. Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1965.
- [83] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1) :42–55, January 1980.
- [84] A. Frommer and D. Szyld. On asynchronous iterations. *J. of computational and applied mathematics*, 23 :201–216, 2000.
- [85] A. Frommer and D. B. Szyld. Asynchronous iterations with flexible communication for linear systems. *Calculateurs Parallèles, Réseaux et Systèmes répartis*, 10 :421–429, 1998.
- [86] Andreas Frommer and Bert Pohl. A comparison result for multisplittings and waveform relaxation methods. *Numerical linear algebra with applications*, 2(4) :335–346, 1995.
- [87] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 1(31) :47–59, January 1982.
- [88] C. W. Gear. The potential for parallelism in ordinary differential equations. Technical Report 1246, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1986.

- [89] C. W. Gear. Massive parallelism across space in ODEs. *Applied Numerical Mathematics : Transactions of IMACS*, 11(1–3) :27–43, January 1993. Parallel methods for ordinary differential equations (Grado, 1991).
- [90] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [91] Bhaskar Ghosh, F. T. Leighton, Bruce M. Maggs, S. Muthukrishnan, C. Greg Plaxton, R. Rajaraman, Andréa W. Richa, Robert E. Tarjan, and David Zuckerman. Tight analyses of two local load balancing algorithms. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 548–558, Las Vegas, Nevada, 29 May–1 June 1995.
- [92] E. Golès, F. Fogelman-Soulie, and D. Pellegrin. Decreasing energy functions as a tool for studying threshold networks. *Disc. Appl. Math.*, 12 :261–277, 1985.
- [93] Duncan Grisby. OmniOrb web page. <http://omniorb.sourceforge.net>.
- [94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message passing interface*. MIT Press, 1994.
- [95] Francesca Guerriero and Roberto Musmanno. Parallel asynchronous algorithms for the K shortest paths problem. *J. Optimization Th. & Appl.*, 104(1) :91–108, January 2000.
- [96] Ernst Hairer and Gerhard Wanner. *Solving ordinary differential equations II : Stiff and differential-algebraic problems*, volume 14 of *Springer series in computational mathematics*, pages 5–8. Springer-Verlag, Berlin, 1991.
- [97] A.V.M. Herz and C.M. Marcus. Distributed dynamics in neural networks. *Physical Review E*, 47(3) :2155–2161, 1993.
- [98] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6) :409–436, 1952.
- [99] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. Nat. Acad. Sci.*, 79 :2554–2558, 1982.
- [100] J.J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Nat. Acad. Sci.*, 81 :3088–3092, 1984.
- [101] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan. Analysis of a graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing*, 10(2) :160–166, October 1990.
- [102] Gregory Karagiorgos and Nikolaos M. Missirlis. Accelerated diffusion algorithms for dynamic load balancing. *Information Processing Letters*, 84(2) :61–67, October 2002.
- [103] P. Koiran. Dynamics of discrete-time, continuous-state hopfield networks. *Neural Computation*, 6 :459–468, 1994.
- [104] V.S. Kozyakin, A. Bhaya, and E. Kaszkurewicz. A global asymptotic stability result for a class of totally asynchronous discrete nonlinear systems. *Mathematics of Control, Signals and Systems*, 12(2) :143–166, 1999.

- [105] M. El Kyal, J.-C. Miellou, and J. M. Bahi. Superlinear convergence of asynchronous waveform relaxation methods for nonlinear ODEs. In *Proceedings of the 9th international colloquium on differential equations*, 1999.
- [106] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45 :255–282, 1950.
- [107] E. Lelarasmee, A. Ruehli, and A. Sangiovanni-Vincentelli. The wavefront relaxation method for time-domain analysis of large scale integrated circuits. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, CAD-1 :131–145, 1982.
- [108] W Lioen, J De Swart, and W Van Der Ween. Test set for ivp solvers. Technical Report NM-R9615, CWI Amsterdam, 1996.
- [109] Boris Lubachevsky and Debasis Mitra. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *Journal of the ACM*, 33(1) :130–150, January 1986.
- [110] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.
- [111] Nicolas Maillard, El Mostafa Daoudi, Pierre Manneback, and Jean-Louis Roch. Contrôle amorti des synchronisations pour le test d’arrêt des méthodes itératives. In *Renpar 14*, pages 177–182, Hamamet, Tunisie, April 2002.
- [112] C.M. Marcus and R.M. Westervelt. Dynamics of analog neural networks with time delay. In D. Touretzky, editor, *Advances in Neural Information Processing Systems I*. Morgan Kauffman, 1989.
- [113] C.M. Marcus and R.M. Westervelt. Dynamics of iterated-map neural networks. *Physical Review A*, 40(1) :501–504, 1989.
- [114] A.N. Michel, J.A. Farrell, and H.-F. Sun. Analysis and synthesis techniques for hopfield type synchronous discrete time neural networks with application to associative memory. *IEEE Transact. Circuits Syst.*, 37(11) :1356–1366, 1990.
- [115] J.-C. Miellou. Algorithmes de relaxation chaotique à retard. *RAIRO, R-1*, pages 52–82, 1975.
- [116] J. C. Miellou, D. El Baz, and P. Spitéri. A new class of asynchronous iterative algorithms with order intervals. *Math. of computation*, 221(67) :237–255, 1998.
- [117] Debasis Mitra. Asynchronous relaxations for the numerical solution of differential equations by parallel processors. *SIAM Journal on Scientific and Statistical Computing*, 8(1) :S43–S58, January 1987. Parallel processing for scientific computing (Norfolk, Va., 1985).
- [118] R. Namyst and J.-F. Méhaut. PM^2 : Parallel multithreaded machine. A computing environment for distributed architectures. In *Parallel Computing : State-of-the-Art and Perspectives, ParCo’95*, volume 11, pages 279–285. Elsevier, North-Holland, 1996.
- [119] D. Pellegrin. *Algorithmique discrète et réseaux d’automates*. PhD thesis, Grenoble, 1986.

- [120] Alan Pope. *The CORBA Reference Guide : Understanding the Common Object Request Broker Architecture*. Addison-Wesley, Reading, MA, USA, December 1997.
- [121] S. P. Rana. A distributed solution to the distributed termination problem. *Information Processing Letters*, 17 :43–46, July 1983.
- [122] Hwakyung Rim, J.-W. Jang, and Sungchun Kim. An efficient dynamic load balancing using the dimension exchange method for balancing of quantized loads on hypercube multiprocessors. In *IPPS/SPDP 1999*, pages 708–713, 1999.
- [123] F. Robert. Théorème de perron-frobenius et stein-rosenberg booléens. *Linear Algebra and Its Applications*, 19 :237–250, 1978.
- [124] F. Robert. *Discrete Iterations, A Metric Study*, volume 6. Springer-Verlag Series in Computational Mathematics, Berlin, 1986.
- [125] F. Robert. *Les Systèmes Dynamiques Discrets*, volume 19. Springer-Verlag, Berlin Heidelberg, 1995.
- [126] Jack L. Rosenfeld. A case study in programming for parallel processors. *Communications of the ACM*, 12(12) :645–655, December 1969.
- [127] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, New York, 1996.
- [128] S. A. Savari and Dimitri P. Bertsekas. Finite termination of asynchronous iterative algorithms. *Parallel Computing*, 22 :39–56, 1996.
- [129] Bruno Scherrer. Parallel asynchronous distributed computations of optimal control in large state space markov decision processes. In *ESANN*, pages 325–330, 2003.
- [130] Yash Shrivastava, Soura Dasgupta, and Sudhakar M. Reddy. Guaranteed convergence in a class of Hopfield networks. *IEEE Transactions on Neural Networks*, 3(6) :951–961, November 1992.
- [131] IEEE Computer Society. Ieee standard for a high performance serial bus. Technical Report Std 1394-1995, IEEE, August 1996.
- [132] Scott D. Stoller. Leader election in asynchronous distributed systems. *IEEE Transactions on Computers*, 49(3) :283–284, 2000.
- [133] D.B. Szyld. Perspectives on asynchronous computations for fluid flow problems. Technical report, Department of Mathematics, Temple University, 2000.
- [134] M. Takeda and J.W. Goodman. Neural networks for computation : Number representations and programming complexity. *Appl. Opt.*, 25(18) :3033–3046, 1986.
- [135] M.N. El Tarazi. Some convergence results for asynchronous algorithms. *Numer. Math.*, 39 :325–340, 1982.
- [136] A. Touzene and B. Plateau. Parallel synchronous and asynchronous iterative methods to solve Markov chain problems. *Supercomputer*, 10(3) :28–39, May 1993.
- [137] Alan Mathison Turing. *Systems of logic based on ordinals : a dissertation*. Ph.D. dissertation, Cambridge University, Cambridge, UK, 1938. Published by Hodgson Son, London, UK.

-
- [138] Paulo Veríssimo and Carlos Almeida. Quasi-synchronism : a step away from the traditional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4) :35–39, 1995.
 - [139] L.P. Wang. On the dynamics of discrete-time, continuous-state hopfield neural networks. *IEEE Trans. Circuits and Systems-II : Analog and Digital Signal Processing*, 45(6) :747–749, 1998.
 - [140] X. Wang, A. Jagota, F. Botelho, and M. Garzon. Absence of cycles in symmetric neural networks. *Neural Computation*, 10 :1235–1249, 1998.
 - [141] Chengzhong Xu and Francis Lau. *Load Balancing in Parallel Computers : Theory and Practice*. 1996.
 - [142] D. M. Young. *Iterative Methods for Solving Partial Difference Equations of Elliptic Type*. PhD thesis, Harvard University, Department of Mathematics, 1950.
 - [143] David M. Young. A historical overview of iterative methods. *Computer Physics Communications*, 53 :1–17, 1989.

Résumé

Dans le domaine du calcul scientifique, les principaux objectifs sont d'effectuer rapidement un très grand nombre de calculs ainsi que de traiter des problèmes de grande taille. Pour atteindre ces buts, le calcul parallèle a été développé, d'abord par le biais de machines spécifiques telles que les calculateurs vectoriels et les machines à mémoire partagée. Puis, avec la baisse des coûts de production des machines et l'augmentation de leur puissance, le parallélisme s'est tourné vers les grappes locales de stations de travail. Enfin, l'amélioration des débits dans le réseau mondial a permis ces dernières années d'envisager l'utilisation de grappes de grappes de machines, c'est-à-dire, des grappes de machines géographiquement distantes et reliées entre elles via l'Internet.

Cette extension du concept de machine parallèle permet théoriquement de rassembler l'ensemble des machines du parc mondial qui sont reliées à l'Internet. Elle a donc le double avantage d'augmenter la puissance de calcul mais aussi la capacité de stockage en mémoire vive et en mémoire de masse, ce qui permet de traiter des problèmes de très grandes tailles.

Cependant, la programmation de telles *méta-grappes* ne peut plus être tout à fait la même que celle des machines classiques ou même des grappes locales de machines. En effet, dans cette nouvelle architecture, l'hétérogénéité des machines utilisées et des liens de communication entre celles-ci génère des contraintes fortes pour obtenir de bonnes performances. Notamment, la synchronisation des communications entre les processeurs induit généralement une perte majeure de performance.

Il est donc essentiel de proposer un nouveau type d'algorithmes qui permette d'utiliser de la manière la plus efficace possible ces méta-grappes. C'est pour ces raisons qu'après avoir travaillé sur des algorithmes parallèles synchrones lors de ma thèse, j'ai orienté mes recherches sur l'asynchronisme, dans le cadre continu d'une part, avec les algorithmes itératifs parallèles, et dans le cadre discret d'autre part, avec les systèmes dynamiques à états finis et temps discret.

L'asynchronisme permet de résoudre efficacement un grand nombre de problèmes scientifiques sur des méta-grappes en apportant la souplesse nécessaire pour s'adapter aux différentes vitesses des processeurs et des liens de communication entre ceux-ci. Néanmoins, pour garantir un comportement stable et obtenir une efficacité optimale, plusieurs aspects doivent être étudiés et/ou adaptés selon le cadre d'utilisation. Dans le cadre continu, on s'intéressera notamment aux conditions de convergence du processus itératif, la détection de cette convergence, la procédure d'arrêt et l'équilibrage de charge. Dans le cadre discret, on s'intéressera également à la convergence du système, en tentant d'identifier précisément les états initiaux qui mènent à un point fixe donné, ainsi qu'à l'étude théorique du comportement asynchrone et à la combinaison du synchronisme et de l'asynchronisme pour concevoir de nouveaux systèmes ayant un comportement donné.

Mots clés : Systèmes dynamiques, asynchronisme, parallélisme, calcul scientifique.

Abstract

In the domain of scientific computing, the main objectives consist in fastly performing a large amount of computations and treating large-size problems. In order to attain those goals, parallel computing has been developed, firstly with dedicated machines such as vectorial computers and shared memory ones. Then, according to the decrease of the production costs of the machines and the increase of their power, parallelism has focused on local clusters. Finally, the enhancement of the throughputs in the worldwide network has allowed to envisage those last years the use of clusters of clusters, that is to say, clusters of geographically distant clusters interconnected via the Internet.

That extension of the concept of parallel machine theoretically permits to gather all the machines in the world which are connected to the Internet. It presents the double advantage to increase the computation power and the memory capacity in RAM and mass storage, which allows the treatment of large-size problems.

However, the programming of such *meta-clusters* cannot be exactly the same as the one of classical machines or even local clusters. Effectively, in that new architecture, the heterogeneity of the machines and of the communication links generates strong constraints to obtain good performances. In particular, the synchronization of the communications between the processors generally induces a major loss of performance.

Thus, it is essential to propose a new kind of algorithms which allows us to use those meta-clusters as efficiently as possible. This is for all those reasons that, after having worked on synchronous parallel algorithms during my PhD Thesis, I have decided to focus my work on asynchronism, in the continuous context on the one hand, with parallel iterative algorithms, and in the discrete context on the other hand, with finite-states discrete-time systems.

Asynchronism permits to efficiently solve a large number of scientific problems on meta-clusters by bringing the necessary flexibility for the adaptation to the different speeds of processors and communication links between them. However, to ensure a stable behavior and to obtain an optimal efficiency, several aspects must be studied and/or adapted according to the context of use. In the continuous case, one will especially focus on the convergence conditions of the iterative process, the detection of that convergence, the halting procedure and the load balancing. In the discrete case, one will also focus on the convergence of the system, by trying to precisely identify the initial states leading to a given fixed point, but also on the theoretical study of the asynchronous behavior and on the mixing of synchronism and asynchronism to design new systems with a given behavior.

Keywords : Dynamical systems, asynchronism, parallelism, scientific computing.