Utilisation des méthodes formelles pour le développement de programmes parallèles

Raphaël Couturier

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

Utilisation des méthodes formelles pour le développement de programmes parallèles

THÈSE

présentée et soutenue publiquement le *24 Janvier 2000*

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Raphaël Couturier

Composition du jury

Rapporteurs :	Jacques JULLIAND	Professeur, Université de Franche Comté, Besançon
	Gérard PADIOU	Professeur, Institut National Polytechnique de Toulouse
	Noëlle CARBONELL	Professeur, Université Henri Poincaré, Nancy I
Examinateurs :	Catherine MONGENET	Professeur, Université Louis Pasteur, Strasbourg I
	Dominique MÉRY	Professeur, Université Henri Poincaré, Nancy I

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Je tiens à remercier Jacques JULLIAND, Gérard PADIOU et Noëlle CARBONELL pour accepter de rapporter mon travail, les examinateurs de cette thèse, Catherine MONGENET et Dominique MÉRY pour m'avoir accompagné dans ce travail, ainsi que Dominique CANSELL.

Je remercie également les personnes avec qui j'ai collaboré :
Enrico CARLON, Christophe CHATELAIN et Christophe CHIPOT.

Je remercie pour finir tous les membres de l'équipe MODEL, les personnes du labo avec qui nous avons passé de bons moments, les deux autres membres de Carcariass ainsi que toutes les personnes que je fréquente en dehors du travail.

À Anne Sophie

Table des matières

Introduction

Le but de ce travail est d'étudier comment les méthodes formelles peuvent être utilisées pour développer des programmes parallèles corrects. Notre objectif est de montrer qu'il est possible de développer des programmes parallèles et que l'on peut paralléliser des programmes séquentiels tout en s'assurant de leur correction. Nous avons appliqué notre approche, dans la mesure du possible, à des simulations numériques.

Contexte

Nous souhaitions travailler sur des applications « réelles », c'est pourquoi une partie de notre travail a été menée en collaboration avec des utilisateurs de calcul intensif non-informaticiens. Durant cette thèse, nous avons réalisé trois expérimentations avec des collègues physiciens ou chimistes. Nous avons effectué ces parallélisations sur les machines du Centre Charles Hermite qui regroupe les utilisateurs de machines parallèles en Lorraine. Il permet ainsi d'établir le dialogue entre les utilisateurs et les futurs utilisateurs de ces machines ayant une culture différente.
Les machines sur lesquelles nous avons travaillé sont des machines à mémoire partagée de SGI (Silicon Graphics Incorporated). La plus performante est l'Origin 2000 qui possède 64 processeurs et 24 Go de mémoire. Par son architecture mémoire physiquement distribuée mais virtuellement partagée, cette machine offre des performances intéressantes en utilisant un paradigme de programmation parallèle avec mémoire partagée ou avec mémoire distribuée. C'est pourquoi, suivant les applications étudiées, nous avons utilisé OpenMP pour la mémoire partagée et MPI pour les applications distribuées. Avec OpenMP, il est possible de paralléliser un programme en décomposant ses données ou les traitements qu'il effectue. On obtient ainsi une décomposition de domaines ou de traitements. L'effort de mise en œuvre et les performances obtenues dépendent de la nature de l'application. Avec MPI, on est contraint de décomposer la structure du programme.

Globalement tous les travaux qui ont pour objectif de développer des programmes parallèles corrects, ont été grandement influencés par les travaux de Hoare [Hoare, 1969] qui définit des programmes dans lesquels il y a des assertions, de Dijkstra [Dijkstra, 1976] qui définit les plus faibles pré-conditions et les commandes gardées pour les programmes non-déterministes. Les travaux d'Owicki-Gries [Owicki et Gries, 1976] permettent de développer de manière formelle un programme parallèle en étendant le travail de Hoare pour les programmes parallèles. Plus tard, Pnueli [Pnueli, 1977] est le premier à utiliser la logique temporelle pour formaliser les propriétés des programmes parallèles. Ensuite, les travaux sur UNITY [Chandy et Misra, 1988] et TLA [Lamport, 1994] définissent chacun une méthodologie utilisée par de nombreuses personnes afin de développer des solutions parallèles.
Ces techniques sont très bien adaptées pour raisonner sur un algorithme. En revanche, elles s'avèrent inadaptées lorsqu'on souhaite les utiliser pour développer une application parallèle en grandeur réelle en raison de la complexité des programmes. La seconde remarque que l'on peut

faire est qu'actuellement il y a peu d'environnements informatiques permettant d'utiliser les travaux cités précédemment pour les programmes parallèles.

Il nous semble qu'il existe deux méthodes intéressantes, pour paralléliser une application en utilisant des outils issus des méthodes formelles. La première développée par Massingill [Massingill, 1998] effectue des transformations sur le code séquentiel en vue d'obtenir une version séquentielle du programme qui simule le comportement du programme parallèle. Ces transformations séquentielles sont guidées par l'utilisation d'archétypes ou patrons qui synthétisent les stratégies de parallélisation par classes de programmes (design patterns). La seconde méthode est élaborée par Sere [Sere, 1990]. Son travail a pour but de développer des programmes parallèles en utilisant le concept de systèmes d'actions. Le formalisme des systèmes d'actions est proche des commandes gardées définies par Dijkstra [Dijkstra, 1976]. Ce formalisme permet de décrire le comportement de programmes parallèles et distribués par l'intermédiaire d'actions que les processus du système exécutent. Plusieurs actions peuvent être exécutées en parallèle tant qu'elles ne partagent pas de variables communes. Ainsi par raffinements successifs guidés par l'objectif visé, nous obtenons un programme parallèle.

Parallèlement, il existe des méthodes qui ne reposent pas directement sur les méthodes formelles mais qui sont intéressantes et qui nous permettent également de situer notre travail. Foster, dans son livre sur le développement de programmes parallèles [Foster, 1995] explique une méthode qui nous semble pertinente pour paralléliser une application. Son approche est pragmatique, elle consiste à découper le problème, ensuite à établir les communications entre les tâches. On regroupe les petites tâches entre elles, enfin on affecte à chaque processeur les processus qu'il doit exécuter. Darlington a élaboré une approche [Darlington *et al.*, 1995] qui consiste à utiliser des *squelettes* pour composer des programmes séquentiels afin d'obtenir des programmes parallèles. Les *squelettes* appliquent les codes séquentiels sur les données qui sont découpées en raison de la parallélisation. L'utilisation des *squelettes* est proche de la programmation fonctionnelle. Cette approche poursuit le travail sur les langages de composition ou de coordination pour le développement de programmes parallèles [Gelernter et Carriero, 1992].

Objectifs et réalisations de cette thèse

Pour paralléliser une application, nous sommes obligés de modifier la structure du programme initial afin que plusieurs tâches puissent s'exécuter de manière concurrente. L'exécution parallèle de plusieurs tâches entraînent un non-déterminisme qui implique une explosion du nombre d'états possible du programme, rendant ainsi la validation par le test très complexe. C'est pourquoi les méthodes formelles semblent un moyen de garantir le développement d'une application parallèle. Suivant le paradigme de parallélisation choisi et suivant le programme étudié, les modifications sur la structure du programme vont être importantes ou non. Généralement, la personne qui parallélise une application se construit un modèle abstrait du programme afin de comprendre schématiquement la structure du programme pour pouvoir proposer un schéma de parallélisation adapté au programme et à l'environnement d'exécution.
Nous avons utilisé la mémoire partagée et la mémoire distribuée comme paradigme de programmation parallèle en raison du contexte applicatif de notre travail. Suivant la nature de l'application, le modèle et la décomposition choisis, il ne nous paraît pas possible d'utiliser une seule stratégie pour développer une application parallèle. C'est pourquoi nous en proposons plusieurs. Sur les expérimentations que nous avons menées, nous nous sommes efforcés à chaque fois d'appliquer la meilleure stratégie.

Dans cette thèse, nous exposons donc les stratégies qui nous paraissent pouvoir répondre à nos attentes, c'est-à-dire garantir que le développement d'une application parallèle satisfasse certains critères de correction. Ces critères dépendent de ce que nous cherchons à établir. Si nous voulons paralléliser une application, nous pouvons montrer que la parallélisation est correcte. Pour effectuer la preuve de propriétés sur un algorithme en vue de développer un programme parallèle, les méthodes formelles apportent l'élément de justification recherché. Finalement, si nous souhaitons obtenir un programme parallèle à partir d'une spécification formelle, il est possible de définir un compilateur ayant ce rôle et nous pouvons définir des règles qui garantissent la traduction.
L'objectif à long terme pour développer une application parallèle est d'établir sa spécification, puis de la prouver et enfin disposer d'un générateur de code qui puisse directement fournir un exécutable pour une machine parallèle. Actuellement, il n'existe pas de tels outils capables de travailler sur des applications réelles. Notre approche consiste donc à utiliser les outils existants et d'essayer de montrer comment on peut, sous certaines hypothèses, apporter des éléments pour contribuer à cette tâche.
Pour les travaux qui nécessitent d'établir des propriétés sur un algorithme ou un programme, nous avons choisi d'utiliser un environnement de développement de preuves appelé PVS, pour *Prototype Verification System* [Crow *et al.*, 1995], car cet outil nous semble bien adapté pour ce travail.
Pour prouver qu'une parallélisation est correcte, nous avons développé une méthode qui utilise un prouveur de théorèmes. Elle est adaptée aux parallélisations utilisant comme paradigme la décomposition de domaines, c'est-à-dire lorsque nous avons réalisé une décomposition des données qui implique une restructuration du programme. Cette méthode demande à l'utilisateur de réaliser une abstraction de son programme. La difficulté est d'élaborer une bonne abstraction, ni trop abstraite, ni trop précise, afin d'obtenir une post-condition du programme. Ensuite nous allons raisonner sur cette abstraction pour montrer que notre parallélisation est correcte. Le but est de montrer que les post-conditions des différentes parties du programme parallèle plus la post-condition du code assemblant ces différentes parties, impliquent la post-condition du programme séquentiel. De manière duale, il est possible de développer un programme parallèle en garantissant sa correction. Pour cela, il faut tenter d'effectuer la preuve précédente sans définir la post-condition du code assemblant les différentes parties parallèles. Certaines obligations de preuves ne sont alors pas prouvables, mais elles constituent la post-condition du code qui assemble les résultats obtenus en parallèle. Ainsi nous pouvons définir, en utilisant les obligations de preuves non prouvées, la post-condition du code qui apporte un résultat correct.
Si au contraire nous parallélisons une application en utilisant le paradigme de mémoire partagée et en effectuant uniquement de la parallélisation de boucles, comme le permet OpenMP, il nous a paru difficile de proposer une méthode pour garantir la parallélisation. La parallélisation est beaucoup plus simple qu'en utilisant une décomposition de domaine, cependant, les problèmes sont à notre avis plus complexes. Les modifications apportées à un programme séquentiel pour le paralléliser ont pour but de supprimer les dépendances entre les itérations des boucles pour que celles-ci soient exécutables en parallèle. Du point de vue du programmeur, ces modifications sont souvent simples. Mais du point de vue preuve, il faut modéliser très finement, l'ordre des exécutions et le déplacement ou la suppression de certaines variables dans une itération n'est pas simple. Les outils de parallélisations automatiques ont pour rôle de transformer automatiquement certaines boucles pour paralléliser un programme en appliquant de tels critères, mais ils ont de nombreuses limites.
Avant d'établir un programme séquentiel ou parallèle, il est possible de définir une spécification du programme qui nous permet de valider certaines propriétés sur celui-ci. Pour montrer que les méthodes formelles sont bénéfiques dans ce cas, nous avons montré que le tri bitonique est

correct. Les propriétés sur ce tri sont prouvées avec PVS et à la fin nous expliquons comment il est possible de paralléliser ce tri. Nous illustrons notre explication par une parallélisation de ce programme en C++.
Finalement, nous avons voulu étudier la possibilité de transformer une spécification formelle en un programme parallèle exécutable sur une architecture parallèle. Pour cela, nous avons choisi de traduire une version d'UNITY adaptée pour le calcul scientifique [Van de Velde, 1994] en un programme Fortran qui utilise la mémoire partagée de l'Origin 2000 grâce aux directives de compilation d'OpenMP. Après avoir montré que notre transformation est valide, nous obtenons un compilateur qui nous permet d'obtenir des performances intéressantes.
Nous avons donc exploré plusieurs moyens de construire des programmes parallèles corrects : en parallélisant un programme séquentiel avec une décomposition de domaines, en élaborant la spécification d'un algorithme, ou en construisant un programme parallèle à partir d'une spécification formelle.

Plan détaillé de cette thèse

Première partie : Dans la première partie nous présentons le cadre de notre travail, le parallélisme, les méthodes formelles et les travaux qui nous permettent de situer notre travail.

Chapitre 1 : Nous présentons dans ce chapitre le parallélisme, les machines parallèles, les difficultés pour développer des applications parallèles, et deux environnements de programmation parallèle. Nous limitons notre présentation compte tenu du contexte de notre travail à OpenMP pour la mémoire partagée et MPI pour les applications où la mémoire est distribuée.

Chapitre 2 : Comment utiliser les méthodes formelles pour développer des programmes parallèles? Nous allons d'abord définir succinctement les différentes catégories de méthodes formelles. Ensuite nous décrivons les deux derniers travaux majeurs pour le développement formel d'algorithmes parallèles. Nous commençons par TLA et ensuite nous nous intéressons à UNITY. Finalement nous étudions PVS, l'environnement de développement de preuves que nous avons utilisé durant tout notre travail.

Chapitre 3 : Nous commençons par présenter les travaux d'Owicki-Gries pour prouver qu'un algorithme parallèle est correct. Nous exposons ensuite deux techniques développées plus récemment pour construire des programmes parallèles en partant de programmes séquentiels. La première, de Massingill, consiste à appliquer des transformations sur le programme séquentiel, à l'aide de modèles appelés archétypes. La seconde approche, de Sere, consiste à modéliser le parallélisme par des systèmes d'actions. Nous décrivons également les méthodes de Foster et de Darlington (qui ne sont pas formelles) pour paralléliser des programmes. Ainsi, nous pouvons situer notre travail.

Deuxième partie : Elle décrit les expérimentations que nous avons menées concernant la parallélisation. Nous avons parallélisé deux applications de physique et une de chimie.

Chapitre 4 : Le premier chapitre de cette partie présente les deux archétypes ou principes que nous avons utilisés pour paralléliser les simulations. Le premier est la décomposition de domaines. Nous l'avons utilisé avec OpenMP et MPI. Le second est la répartition de traitements que nous utilisons avec OpenMP en parallélisant les boucles d'un programme.

Chapitre 5 : Il décrit la parallélisation d'une simulation d'un système de transition de phase, réalisé par un physicien de Nancy I. La parallélisation est basée sur une décomposition de domaines. Dans ce cas, nous avons utilisé OpenMP pour paralléliser les boucles du programme, puis nous avons utilisé MPI. Une seule partie du programme pose des difficultés pour la parallélisation.

Chapitre 6 : Ce chapitre est consacré à la parallélisation d'une simulation d'un système de dynamique moléculaire. Cosmos est un programme utilisé par un chimiste de Nancy I. Il a été parallélisé avec OpenMP. Ce chapitre montre qu'OpenMP se révèle un moyen efficace de paralléliser une application sans la connaître entièrement.

Chapitre 7 : Il décrit une autre simulation d'un système de transition de phase parallélisée avec MPI. Cette simulation nous a permis de tester l'utilisation d'OpenMP en plus de MPI. Nous verrons que les résultats sont encourageants.

Troisième partie : Cette partie présente les stratégies que nous proposons pour développer des programmes parallèles. Notre première parallélisation nous a permis de développer une technique pour prouver qu'une application est correctement parallélisée et pour trouver la condition qui fera qu'une parallélisation est correcte pour la décomposition de domaines. Nous avons utilisé PVS [Crow *et al.*, 1995] pour construire une spécification d'un tri à partir des propriétés que nous avons prouvées. Enfin, nous avons construit un compilateur pour générer un programme parallèle à partir d'une spécification UNITY.

Chapitre 8 : Dans ce chapitre, notre but est de montrer que nons pouvons utiliser les méthodes formelles pour créer la spécification d'un algorithme. Nous avons choisi le tri bitonique. Les propriétés que notre algorithme doit respecter sont énoncées puis, nous devons prouver si elles sont correctes, sinon nous devons les corriger. Finalement, lorsque toutes les propriétés sont vérifiées, nous pouvons construire notre spécification. Celle-ci est assez proche d'un programme parallèle exécutable, donc nous pouvons la traduire facilement en C++.

Chapitre 9 : Nous avons développé une méthodologie afin de prouver qu'une parallélisation utilisant la décomposition de domaines s'avère correcte. Ce chapitre explique comment il faut procéder pour prouver qu'une telle parallélisation est correcte. Nous devons prouver que les post-conditions des différentes parties du programme parallèle, plus la post-condition de la partie réalisant le collage, impliquent la post-condition du programme séquentiel. De plus, nous pouvons faire la même preuve sans la post-condition du code qui assemble les résultats et, dans ce cas, les obligations de preuves constituent la post-condition de ce code. Cette méthodologie a été appliquée aux deux parallélisations basées sur une décomposition de domaines (chapitres 5 et 7).

Chapitre 10 : Dans ce chapitre, nous expliquons comment nous avons construit un compilateur pour transformer une spécification formelle écrite en UNITY parallèle en un programme Fortran utilisant OpenMP qui fonctionne sur une machine à mémoire partagée. Nous avons travaillé avec une version d'UNITY proposée par [Van de Velde, 1994]. L'intérêt est à long terme de proposer une méthode de preuves d'applications parallèles qui puissent générer du code exécutable efficace pour ne pas être obligé de le générer à la main, ce qui peut encore être source d'erreurs.

Conclusions et perspectives : Nous donnons les conclusions et les perspectives de ce travail.

Publications issues de cette thèse :

- Raphaël Couturier et Dominique Méry, *Coordination of abstract machines.* Dans CSIT'97, Yerevan, Arménie, Septembre 1997.
- Raphaël Couturier, *Parallélisation d'une simulation Monte Carlo d'un système de spins et preuve.* Dans RENPAR'10, Strasbourg, France, Juin 1998.
- Raphaël Couturier et Dominique Méry, *An experiment in parallelizing an application using formal methods.* Dans CAV'98, LNCS, Vancouver, Canada, Juin 1998.
- Raphaël Couturier, *Formal engineering of the bitonic sort using PVS.* Dans IWFM'98, Cork, Irlande, Juillet 1998.
- Raphaël Couturier et Dominique Méry, *Parallelization of a Monte Carlo simulation of a spins system.* Dans PDPTA'98, Las Vegas, USA, Juillet 1998.
- Raphaël Couturier, Bertrand Couturier et Dominique Méry, *A compiler for parallel Unity programs using OpenMP.* Dans PDPTA'99, Las Vegas, USA, Juillet 1999.
- Raphaël Couturier et Christophe Chipot, *Parallel molecular dynamics using OpenMP on a shared memory machine.* Computer Physics Communications. Vol. 124, No. 1, pages 49-59, Janvier 2000.
- Raphaël Couturier, *Trois expérimentations de simulations parallèles.* Accepté à TSI pour le thème : Simulation et calcul haute performance.

Première partie

Présentation des techniques utilisées

1

Parallélisme

Ce chapitre présente les notions sur le parallélisme indispensables pour comprendre le travail que nous avons effectué. Disposant de plusieurs modèles, nous présentons plus en détail ceux que nous avons utilisés pour nos travaux. Nous détaillons l'utilisation d'OpenMP et de MPI. Nous comparons ces deux modèles de programmation parallèle. Nous décrivons également le Centre Charles Hermite (CCH) qui nous a permis de mener ce travail.

1.1 Pourquoi le parallélisme et quel parallélisme?

On utilise le parallélisme afin de pouvoir exécuter un travail plus rapidement ou de pouvoir exécuter un travail plus important, en utilisant plus de ressources, dans le même temps.
Il existe différentes catégories de parallélisme. On peut différencier le parallélisme de données, le parallélisme de contrôle, le parallélisme des entrées et sorties. Le premier partage les données entre les processeurs et ainsi chaque processeur traite seulement une partie des données. Pour le second type, le travail réalisé par un seul processus va être décomposé afin que plusieurs processus, s'exécutant sur plusieurs processeurs, puissent réaliser un travail moins important. Finalement, la troisième catégorie se distingue par la possibilité de pouvoir lire et / ou écriture simultanément sur un ou plusieurs média.
Dans ce document, nous nous intéressons plus particulièrement aux deux premières catégories de parallélisme.
Pour le parallélisme de données ou de contrôle, suivant la manière de découper les données ou de répartir le contrôle, nous obtenons encore deux classes générales de parallélisme. Classiquement, on distingue le parallélisme à gros grain, du parallélisme à grain fin . Si le découpage est grossier, on obtient du parallélisme à gros grains. Par exemple, concernant le parallélisme de données, on peut découper une grille de n^2 points en 16 sous-grilles de $(n/4)^2$ points. De même, pour le parallélisme de contrôle, on peut découper un traitement de n itérations en 16 traitements de $n/16$ itérations. Si le découpage est plus fin, on obtient du parallélisme à grain fin. Dans ce cas, en reprenant les deux exemples précédents, on peut considérer par exemple la grille comme un ensemble de n^2 points dans le cas du parallélisme de données et on a la possibilité de considérer le traitement initial comme n traitements dans le cas du parallélisme de contrôle.

1.2 Quelles machines?

On distingue habituellement trois catégories de machines parallèles. La première est constituée des machines à l'intérieur desquelles tous les processeurs peuvent accéder à la même mémoire,

d'où le nom de machines à mémoire partagée. Avec cette architecture, tous les processeurs peuvent écrire dans la mémoire simultanément. Mais si on souhaite obtenir des résultats cohérents, il faut garantir l'ordre d'écriture de chaque mot mémoire. Ceci constitue souvent une contrainte difficile à respecter. Par contre, l'énorme avantage de cette architecture réside dans le fait que tous les processeurs peuvent lire simultanément[1] la même information contenue dans la mémoire.
La seconde architecture est constituée d'un ensemble de processeurs disposant chacun d'une mémoire locale. Dans ce cas, il y a encore deux architectures différentes. On distingue le cas où les processeurs s'échangent des informations par l'intermédiaire de l'envoi de messages. Donc le problème de l'accès en écriture concurrent n'existe plus mais la difficulté réside dans l'obligation de spécifier explicitement l'envoi de messages. La seconde architecture permet à un processeur d'écrire ou de lire dans la mémoire d'un processeur distant. Il est pour l'instant moins répandu en raison de la nécessité d'un matériel spécifique.
En plus de ces types fondamentaux d'architectures, il existe des modèles hybrides qui sont en fait un mélange de celles-ci. On distingue alors les machines à mémoire distribuée et virtuellement partagée des réseaux de stations multiprocesseurs à mémoire partagée. Les machines de la première catégorie sont appelées *NUMA* pour *Non-Uniform Memory Access* puisque l'accès à la mémoire n'est pas uniforme et dépend de la distance entre le processeur et la mémoire. Si en plus il existe un système de cohérence de cache, l'architecture est appelé *ccNUMA* pour *cache coherent NUMA*. L'Origin 2000 est un bel exemple de *ccNUMA* car la mémoire sur cette machine est physiquement distribuée mais virtuellement partagée. Ainsi, le système et les composants matériels qui offrent la cohérence de cache, masquent l'architecture qui est distribuée pour des raisons de performances et de coût, et donnent l'impression d'avoir une machine à mémoire partagée « classique ». Pour la seconde catégorie, les principaux vendeurs de stations de travail proposent maintenant des versions multiprocesseurs à mémoire partagée de leur stations de travail.

1.2.1 Présentation des machines du Centre Charles Hermite

Nous allons détailler les machines parallèles que nous avons utilisées essentiellement dans cette thèse. En août 1999, ce centre comprenait 4 Power Challenge Array comportant de 4 à 16 processeurs, cadencés à 195Mhz, avec en moyenne 1Go de mémoire. Ces machines possèdent une mémoire partagée. Les processeurs accèdent à la mémoire centrale par l'intermédiaire d'un bus spécialisé à cet effet. Malheureusement, dans la pratique, ce bus limite généralement les performances de la machine. Les Power Challenge représentent les machines de la précédente génération de SGI (Silicon Graphics Incorporated).
L'Origin 2000 est issue de la nouvelle génération de machines SGI. Elle possède une architecture distribuée et virtuellement partagée. Ainsi, l'utilisateur a l'impression de programmer une machine à mémoire partagée. Ce sont les composants matériels qui prennent en charge le transit des données lorsqu'il est nécessaire. L'Origin 2000 du centre possède 64 processeurs, cadencés à 195Mhz, avec 24Go de mémoire. C'est sur cette machine que nous avons travaillé le plus, en raison de sa rapidité, de sa souplesse et de son attractivité. Dernièrement, le centre a fait l'acquisition d'1To de disque dur (nous ne nous sommes pas intéressés à cette particularité) et d'une Origin 2000 spécialisée pour le graphique qui dispose de 4 processeurs, cadencés à 250Mhz.
Afin de contrôler les processus fournis à ces deux catégories de machines, il y a deux Origin 200 couplées, donc une machine disposant de 4 processeurs cadencés à 180Mhz qui utilise NQE[2]

1. Même si dans la pratique la lecture est réalisée pour le moment de manière séquentielle à quelques cycles machines d'intervalle. L'ordre de lecture n'a pas d'importance tant qu'il n'y a pas d'écriture. De plus, la probabilité pour que des processeurs aient besoin de la même donnée au même top d'horloge est rare.

2. NQE est un produit développé par SGI pour gérer la soumission et l'exécution des processus d'une machine

pour effectuer ce contrôle. Cette machine sert également de machine de mise au point pour les programmes en développement que l'on doit tester.

1.3 Quels sont les problèmes rencontrés pour développer des applications parallèles ?

Pour développer une application parallèle, on rencontre déjà tous les problèmes liés au développement de logiciels séquentiels, et en plus des problèmes spécifiques au parallélisme. Nous énumérons rapidement et non exhaustivement les problèmes liés au développement en séquentiel :

- La mise au point et la maintenance consistent à modifier le programme afin qu'il permette de satisfaire les demandes du client. Ces phases sont parfois longues et elles nécessitent parfois de grosses modifications structurelles.
- La portabilité a pour but de pouvoir exécuter le même programme sur le maximum d'architectures et d'environnements systèmes possible, sans toutefois tout récrire à chaque fois.
- L'évolution d'un programme est indispensable car en l'utilisant on se rend compte qu'il faudrait ajouter de nouvelles fonctionnalités au programme.
- La phase de test permet de certifier qu'un programme aura un comportement correct, mais faire un test exhaustif est souvent difficile voire même impossible.
- La correction d'un programme est trop peu souvent entreprise mais elle permet néanmoins, en utilisant des outils dérivés des méthodes formelles, de prouver qu'un programme est correct.

Voici la liste des problèmes spécifiques au développement de programmes parallèles :

- Le problème fondamental est qu'un programme parallèle engendre du non-déterminisme et une explosion combinatoire des exécutions possibles. À cela s'ajoute la non reproductabilité d'une exécution en raison de l'ordonnancement des processus qui varie souvent. C'est pourquoi la certification par le test devient encore plus difficile que pour un programme séquentiel.
- Avant de vérifier si les résultats sont corrects, il faut tout d'abord s'assurer que le programme ne sera pas bloqué. Les causes de blocages sont multiples, par exemple deux processus attendant mutuellement le résultat de l'autre processus sont bloqués et empêchent le programme de se terminer.
- Pour la mise au point, il existe de nombreux *débogueurs* pour les programmes séquentiels ; mais concernant les programmes parallèles, suivant le paradigme de parallélisation utilisé, le langage et l'architecture utilisés, les *débogueurs* performants sont plutôt rares.
- Les accès aux variables ainsi qu'aux ressources doivent être gérés de manière précise. Si par exemple deux processus veulent modifier simultanément la même donnée, ce n'est pas possible, donc il faut établir un ordre d'écriture qui garantit le résultat final.
- Lorsqu'on parallélise un programme séquentiel, ce qui est souvent le cas, il est important de savoir si on obtiendra le même résultat avec le programme parallèle qu'avec le programme séquentiel.
- La parallélisation doit nécessairement apporter un gain « important » sinon l'investissement en machine et en temps pour développer le programme parallèle ne présente aucun intérêt. Cela paraît être une évidence, mais dans les faits, des gains appréciables ne sont pas toujours faciles à obtenir.

parallèle.

- L'absence de standard entre les machines et les outils fait qu'il est difficile de choisir l'architecture et l'environnement sur lesquels vont s'exécuter un programme. Ce choix est souvent crucial car le passage d'une plateforme à une autre est difficile et coûteux.

1.4 Environnements de programmation parallèle utilisés

Le développement de systèmes pour programmer une grande variété d'applications a toujours été un grand souci. Concernant les machines parallèles, il existe une multitude de paradigmes permettant de les programmer. Nous allons voir en détail les deux environnements de programmation parallèle que nous avons utilisés. OpenMP est une implantation du paradigme de programmation parallèle par mémoire partagée. Il n'est utilisable que sur les machines à mémoire partagée. Le second, MPI, a été développé initialement pour les machines à mémoire distribuée. Sa portabilité fait qu'on le retrouve également sur les architectures à mémoire partagée.

1.4.1 OpenMP

L'apparition des machines à mémoire partagée et surtout la popularité de ce type de machines avec plus d'une vingtaine de processeurs, a fait émerger le besoin de développer une bibliothèque dédiée à cette architecture. Les principaux constructeurs et développeurs se sont réunis, afin de définir une norme : OpenMP [Open consortium, 1997b]. Cette norme a pour but d'utiliser, de la manière la plus efficace possible, les machines SSMP (Scalable Shared Memory multiProcessors pour « multi-processeurs à mémoire partagée extensible »). Elle est la synthèse de plusieurs systèmes développés par plusieurs constructeurs pour développer des applications parallèles avec directives. Le principe est le suivant : on introduit des directives dans un programme séquentiel et celles-ci aident le compilateur à construire un code parallèle. Actuellement, OpenMP est disponible avec les compilateurs Fortran 77, Fortran 90, C et C++. Mais c'est avec Fortran qu'il se révèle le plus efficace car, de manière générale, c'est avec ce langage que les compilateurs peuvent effectuer le plus d'optimisations en raison de sa construction plus élémentaire à la base.

Modèle sous-jacent

Le modèle d'exécution d'OpenMP [Open consortium, 1997a] est défini de la manière suivante : tout d'abord, OpenMP commence son exécution avec un seul processus que l'on appelle le processus maître (ou en anglais *thread* en raison de la différence avec les processus lourds hérités d'unix[3]). Ensuite le *thread* maître s'exécute en séquentiel jusqu'à ce qu'il atteigne une construction parallèle. Dans les faits, on en distingue deux. Si on utilise le parallélisme à grain fin, on parallélise la plupart des boucles de l'application (quelquefois les boucles sont très grosses et donc le terme de parallélisation à grain fin n'est peut-être pas approprié). Si on considère le parallélisme à gros grain, on définit dans le code des sections parallèles qui sont exécutées par tous les *threads* mis à disposition par l'utilisateur (dans ce cas il n'y a aucune différence sémantique avec le concept de la programmation parallèle SPMD (Single Program Multiple Data pour « programme unique à données multiples ») basée sur les bibliothèques à envoi de messages dans le sens où chaque *thread* exécute le même programme et des calculs sur des données différentes suivant son numéro). Donc, dès qu'il rencontre une construction parallèle, le *thread* maître répartit le travail entre les différents *threads*, il s'occupe de la synchronisation et continue à s'exécuter en séquentiel après l'exécution de la construction parallèle.

3. Dans la suite du document on distinguera les *threads*, processus légers, des processus classiques d'Unix.

Les travaux que nous avons parallélisés avec OpenMP utilisent le paradigme du parallélisme à grain fin car à chaque fois, le code séquentiel existait. Dans ce cas, il est plus facile de procéder de la sorte, surtout si nous n'avons pas conçu le programme séquentiel. Nous allons détailler la démarche qu'un utilisateur doit suivre pour paralléliser les boucles d'une application avec OpenMP.

Principes d'utilisation et fonctionnement

Pour savoir si une boucle peut être parallélisée, il faut savoir si toutes les itérations de la boucle peuvent s'exécuter en parallèle, c'est-à-dire qu'il ne doit pas exister de dépendances entre les itérations. Il existe des outils qui permettent d'analyser les dépendances entre les itérations. On peut citer par exemple TransTool [Darte *et al.*, 1996]. Les compilateurs[4] et les outils capables de faire de la parallélisation automatique intègre ce genre d'analyse. Lorsqu'une boucle, se prêtant à la parallélisation est identifiée, il faut spécifier au compilateur le statut des variables intervenant dans la boucle. Il y a trois types de cas possibles. Soit la variable est *PRIVATE* et, dans ce cas, chaque processeur possède une copie de la variable qu'il peut modifier librement. On utilise ce type de variable pour tous les compteurs de boucles et les variables scalaires locales à une boucle (même si la notion de variable locale n'a pas de sens en Fortran). Soit la variable est *SHARED* et dans ce cas, la variable est partagée entre tous les processus. On utilise ce type de variable pour les variables scalaires auxquelles on accède en lecture seulement et pour les tableaux. Si on accède à un élément d'un tableau en lecture et / ou écriture par au moins deux processus, le travail du programmeur consiste à supprimer l'interaction en créant, la plupart du temps, un nouveau tableau avec une dimension supplémentaire. Cette nouvelle dimension permet à chaque processeur d'effectuer ses calculs. Ensuite, nous collectons les différents résultats calculés par chaque processeur. Il faut noter que par défaut, si on ne spécifie rien, une variable possède le statut *SHARED*. Le dernier type de variable que l'on peut spécifier est *REDUCTION*. On utilise ce type de variable lorsqu'une variable est modifiée par une seule opération simple (+, -, *, /) par toutes les itérations. Dans ce cas, chaque processeur calcule un résultat partiel dans une variable locale et à la fin de la boucle, on applique la même opération entre tous les résultats partiels pour obtenir le résultat final.
Dès que l'on a spécifié le statut de chaque variable, le compilateur génère un code qui distribue, dynamiquement les variables suivant leur statut et les itérations des boucles, suivant le nombre de processeurs alloués. Par exemple, si une boucle possède 1000 itérations et que nous disposons de 10 *threads*, alors le *thread* maître, numéro 0, va exécuter les itérations 1 à 100, le *thread* numéro 1 va exécuter les itérations 101 à 200, et ainsi de suite (on peut également utiliser d'autres politiques pour la distribution des itérations). On peut constater une certaine similitude Bien qu'il n'y ait aucune relation de type sémantique entre OpenMP et le parallélisme de données , on peut tout de même constater une certaine similitude entre ces deux concepts. Le parallélisme de données possède la particularité suivante : tous les processeurs exécutent les mêmes instructions, en parallèle, avec des données différentes. Avec OpenMP, chaque processeur exécute une itération différente de la même boucle, donc seules les données diffèrent entre les processeurs, les instructions restent les mêmes.
Un avantage d'OpenMP et des langages parallèles avec directives réside dans le fait que ces directives peuvent être ignorées par les compilateurs. Ainsi on obtient un programme séquentiel. C'est très utile car, avant de paralléliser une boucle, il faut supprimer les dépendances et donc il faut s'assurer que le programme séquentiel est toujours correct. Le travail que l'on vient de

4. Le compilateur SGI peut nous fournir un rapport sur les dépendances des itérations des boucles d'un programme.

décrire peut être en partie réalisé par des outils de parallélisation de boucles. En général, tous les outils qui réalisent de la parallélisation automatique présentent des difficultés lorsque les bornes des compteurs des boucles et les indices des tableaux ne sont pas des expressions affines des compteurs extérieurs et de constantes entières.

Un petit exemple

Voyons maintenant sur des exemples simples comment on place les directives quand c'est possible avec la figure 1.1. En annexe A.1.1, nous donnons un exemple de multiplication d'une matrice par un vecteur en utilisant OpenMP.

```
DO i=1,n                DO i=1,n                DO i=1,n
  k=A(i)                  k=A(i)                  k=A(i)
  DO j=1,k                DO j=1,k                DO j=1,k
    B(i)=k*B(i)+C(i)        B(k)=k*B(k)+C(k)        B(j)=k*B(j)+C(j)
  ENDDO                   ENDDO                   ENDDO
ENDDO                   ENDDO                   ENDDO
```

boucle 1 *boucle 2* *boucle 3*

FIG. 1.1 – *3 différentes boucles que l'on veut paralléliser*

Dans la *boucle 1*, chaque itération `i` a une valeur `k` utilisée pour calculer la boucle imbriquée sur `j` qui met à jour la valeur de `B(i)`. Donc, toutes les itérations sont indépendantes les unes des autres. On peut ainsi exécuter toutes les itérations de manière concurrente si `k` est privé.
Dans la *boucle 2*, chaque itération `i` a une valeur `k` utilisée pour calculer la valeur `B(k)` de la boucle imbriquée sur `j`. On peut distinguer deux cas suivant les valeurs de `A`. Si on sait que toutes les valeurs de `A` sont différentes, toutes les itérations peuvent être exécutées en parallèle car chaque itération modifie une valeur de `B` différente. Dans le cas contraire, on ne peut pas exécuter les itérations en parallèle et garantir le même résultat qu'en séquentiel.
Dans la *boucle 3*, chaque itération `i` a une valeur `k` qui est utilisée pour mettre à jour la valeur de `B(j)` dans la boucle imbriquée `j`. Donc, certains éléments de `B` sont susceptibles d'être mis à jour par plusieurs itérations `i`, c'est pourquoi on ne peut pas exécuter ces itérations `i` de manière concurrente. Donc cette boucle n'est pas parallélisable.
La figure 1.2 montre comment nous avons inséré les directives de compilation dans le code. On ne donne pas de version parallèle pour la *boucle 3* car nous avons précisé qu'elle n'était pas parallélisable.

1.4.2 L'envoi de messages et MPI

Les architectures

Actuellement les machines à mémoires distribuées sont sûrement, pour des raisons de coût, les machines les plus couramment utilisées. Concernant ce type de machine, on peut distinguer deux grandes catégories. La première est constituée des machines dédiées au calcul parallèle. Dans ce type d'architecture, chaque processeur possède une mémoire locale et les connexions entre les différents processeurs sont assurées par un réseau spécialisé. La plupart du temps, les constructeurs choisissent une topologie particulière (on peut citer par exemple les grilles 2D ou 3D, les hyper-cubes ou les tores). La seconde catégorie, appelée *cluster* de stations, regroupe toutes les stations de travail « classiques » reliées par un réseau éthernet rapide, mais celles-ci

```
!$OMP  PARALLEL DO PRIVATE(i,k,j)
!$OMP+ SHARED(A,B,C)
DO i=1,n
  k=A(i)
  DO j=1,k
    B(i)=k*B(i)+C(i)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

boucle 1

```
!$OMP  PARALLEL DO PRIVATE(i,k,j)
!$OMP+ SHARED(A,B,C)
DO i=1,n
  k=A(i)
  DO j=1,k
    B(k)=k*B(k)+C(k)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```

boucle 2

FIG. 1.2 – *les boucles 1 et 2 parallélisées, la boucle 2 donne un résultat correct seulement si tous les éléments de A sont distincts.*

sont loin d'offrir les mêmes performances que les réseaux de machines parallèles à mémoires distribuées. Pour cette seconde catégorie d'architecture, chaque processeur dispose également d'une mémoire qui lui est propre puisqu'on a des stations de travail. Le réseau est souvent un point faible comparé au réseau des architectures à mémoire partagée dédiées au calcul parallèle. D'où l'apparition de *clusters* de stations reliées par un réseau spécialisé. Certains réseaux sont composés d'un *switch*, qui aiguille les messages entre les différentes machines. Actuellement, avec les *clusters* de stations, les machines sont séparées au maximum d'une longeur de trois mètres avec la technologie Myrinet. Avec des PC, le problème essentiel vient du fait que le bus PCI est susceptible de constituer un goulot d'étranglement car les données peuvent arriver plus rapidement sur la carte réseau que le bus PCI n'est capable de les traiter.
Comme chaque processeur dispose d'une mémoire locale et qu'il ne peut pas accéder directement à la mémoire des autres processeurs, il est nécessaire que les processeurs s'envoient des messsages contenant des données pour pouvoir réaliser un travail en parallèle. De nombreuses bibliothèques ont été réalisées, mais les deux plus courantes sont PVM [Geist *et al.*, 1994] et MPI [Gropp *et al.*, 1994]. Pour le moment, MPI est plus populaire. Il existe cependant des modèles qui ont pour but de cacher l'architecture distribuée. On peut citer Linda [Donald et Berndt, 1989] qui, à partir de 4 commandes de base, simule une mémoire partagée. Une version utilisant C++ a été développée dans notre équipe [Galibert, 1997].

Le principe

Maintenant nous pouvons détailler le fonctionnement des bibliothèques à envoi de messages. Faisons une analogie avec un message postal : pour qu'un message véhicule, il lui faut un expéditeur, un destinataire et il faut préciser la nature et le poids de celui-ci. Pour qu'un message véhicule des informations entre des processeurs, il faut également un processeur expéditeur et un récepteur ; on doit aussi préciser le contenu et la taille de ce message. La manière dont il est transporté n'importe pas plus au concepteur d'une application parallèle que le moyen utilisé pour transporter une lettre n'importe à la personne qui l'a écrite. Il existe plusieurs manières d'envoyer un message à un processeur et plusieurs manières de recevoir un message, suivant que l'on souhaite que les communications soient synchrones ou asynchrones et bloquantes ou non.
Lorsque les communications sont synchrones, l'émetteur et le récepteur se synchronisent pendant l'échange du message. Ce type d'échange est dit par rendez-vous. Les primitives d'envoi et de réception sont bloquantes et elles se terminent sensiblement au même instant.
Lorsque les communications sont asynchrones, l'émission et la réception ne se terminent pas au même instant. Il est même possible que l'une se termine avant que l'autre ait commencé.

Suivant si les communications sont bloquantes ou non, on distingue trois possibilités : envoi et réception non bloquants, envoi bloquant et réception non bloquante, envoi non bloquant et réception bloquante [Dillon, 1997].
L'analogie que nous venons de faire avec un message postal convient très bien pour les communications point à point, c'est-à-dire entre deux processeurs, mais elle se révèle inadaptée pour des communications dites collectives. Pour ce type de communication, on souhaite qu'une partie, voire la totalité des processeurs s'échangent une ou des informations. Par exemple, tous les processeurs envoient un résultat à un processeur qui se charge de la centralisation, ou tous les processeurs envoient à tous les autres processeurs, une partie d'un résultat permettant de calculer la suite des résultats. Il existe de très nombreux moyens de communiquer pour chaque bibliothèque [Gropp *et al.*, 1994; Geist *et al.*, 1994].

MPI (Message Passing Interface)

MPI est la bibliothèque de communication la plus utilisée en raison de son succès auprès des utilisateurs tout comme auprès des constructeurs. Il existe de très nombreuses implantations de MPI suivant qu'elle fonctionne sur une architecture dédiée ou non. De même, suivant les implantations, toutes les fonctions courantes de MPI existent, mais certaines fonctions moins courantes n'existent pas ou sont implantées de manière non optimale à partir d'autres primitives. Le principe de base est que dans la version actuelle (mais cela changera un jour ou l'autre), la création des processus est statique, contrairement à PVM par exemple. C'est-à-dire qu'il faut préciser le nombre de processus au démarrage de l'application. Ce nombre ne peut pas changer, malheureusement pour certaines applications de nature plus dynamique. On peut également spécifier au démarrage quel programme sera exécuté par quel processeur. On identifie chaque processus par un numéro et l'envoi d'un message consiste à nommer le processeur qui enverra le message ainsi que celui qui le réceptionnera. Les communications s'effectuent à travers des canaux logiques (qui n'ont aucun support physique). Ainsi on peut restreindre la circulation des messages uniquement aux processeurs appartenant à un même canal. Il existe tous les types de communication classique point à point et collective [Gropp *et al.*, 1994]. En annexe A.1.2, nous donnons un exemple de multiplication de matrice par un vecteur utilisant MPI.

1.5 Avantages et inconvénients d'OpenMP et MPI

Il peut s'avérer utile, avant de choisir tel ou tel paradigme de programmation, de connaître les avantages et les inconvénients de chacun afin de choisir le meilleur environnement suivant les choix dont on dispose.
Les arguments que nous avançons sont généraux, cependant nous préciserons pour certaines architectures les changements notables.

1.5.1 Avantages et inconvénients d'OpenMP

Commençons tout d'abord par les avantages :

- Comme nous l'avons décrit, l'utilisation d'OpenMP est assez simple et elle permet la mise en œuvre d'une parallélisation assez rapidement.
- La programmation est de haut niveau car on ne spécifie pas directement les communications.
- La transition entre le code séquentiel et le code parallèle est aisée car il suffit de recompiler le programme en spécifiant au compilateur s'il doit ignorer ou en tenir compte des directives.

Les inconvénients sont les suivants :

- Le parallélisme à grain fin peut se révèler inefficace lorsque le nombre de processeurs utilisés est élevé. Pour les machines à mémoire réellement partagée, le nombre de processeurs reste pour le moment relativement limité. Pour les machines à mémoire distribuée et virtuellement partagée, lorsque le nombre de processeurs utilisés devient important, le temps de distribution des itérations et des données devient important alors que le temps d'exécution des calculs de chaque processeur diminue.
- Pour l'instant, peu de bibliothèques ont été développées spécifiquement pour ce modèle, en raison de sa jeunesse et de son utilisation restreinte à certaines machines.
- La conception d'une application utilisant le parallélisme à gros grain demande le même effort de découpage des calculs que si on utilise une bibliothèque à envoi de messages, toutefois les communications n'ont pas besoin d'être explicitées.

1.5.2 Avantages et inconvénients de MPI

Les avantages de MPI sont les suivants :

- Il n'y a pas de problème de localité de données car toutes les données sont locales.
- Un code développé avec MPI est censé être portable, néanmoins suivant les fonctions qu'on utilise, on rencontre quelquefois des problèmes car certains constructeurs de machines n'implantent qu'une partie des fonctions de MPI.
- En raison de sa portabilité, il existe de nombreuses bibliothèques de calculs déjà parallélisées utilisant MPI ainsi que de nombreux codes.

Les inconvénients de MPI sont les suivants :

- Le découpage d'une application, afin que plusieurs processus puissent l'exécuter, est souvent loin d'être trivial.
- Il faut minimiser les communications afin d'être rapide et cette tâche n'est pas aisée.
- La mise en oeuvre est souvent difficile et longue en raison des concepts qui sont de bas niveau.

1.6 Conclusion

Les machines parallèles se distinguent suivant la mémoire qu'elles possèdent : distribuée ou partagée. Ces deux catégories de machines sont à l'origine des bibliothèques permettant de paralléliser des calculs. Nous utilisons le paradigme d'envoi de messages avec MPI et la mémoire partagée avec OpenMP. Ces deux environnements de programmation parallèle sont très différents et ils ont chacun des caractéristiques intéressantes. MPI est très portable, mais n'est pas toujours facile à utiliser, surtout en raison du bas niveau de programmation qu'il procure. OpenMP est par contre plus facile à utiliser, mais son utilisation est limitée aux machines à mémoire partagée. Suivant la nature de l'application, du niveau de programmation que l'on désire et de l'architecture visée, nous allons utiliser l'un ou l'autre de ces deux environnements, voire même les deux. La construction d'applications parallèles est souvent effectuée à partir d'une version séquentielle. Pour garantir que la parallélisation d'une application séquentielle est correcte, nous allons essayer d'utiliser les méthodes formelles.

2

Méthodes formelles

Ce chapitre présente les méthodes formelles de manière générale tout en détaillant TLA, UNITY qui permettent de spécifier des systèmes concurrents. Nous choisissons de présenter TLA, en raison de la vue plus générale qu'il procure par rapport à l'équité. Les spécifications TLA sont plus abstraites que les spécifications UNITY, ces dernières étant plus proche d'un langage de programmation. Pour ces deux méthodologies, fondées sur la logique temporelle, nous donnons succinctement les règles qui permettent de prouver des propriétés de programmes. Le reproche que nous pouvons faire à ces deux méthodologies est qu'il n'y a pas d'outil satisfaisant pour mécaniser des preuves avec elles. Par ailleurs, nous cherchons plus à montrer qu'un programme parallélisé va donner des résultats corrects, qu'à montrer qu'il n'y aura pas de problèmes d'allocation de ressources. C'est pourquoi nous détaillons également l'utilisation de PVS, un environnement de développement de preuves utilisant la logique classique. Nous avons utilisé fréquemment cet outil pour construire des preuves en raison de sa puissance.

2.1 Présentation des méthodes formelles

2.1.1 Pourquoi les méthodes formelles?

La conception d'un programme est une processus long qui nécessite plusieurs étapes. Tout d'abord, il faut établir un cahier des charges afin de définir de manière informelle les buts et les contraintes du programme. La phase de conception définit les structures de données, les modules et les relations entre ces différentes entités. L'objectif est d'implanter le programme afin qu'il réalise ce qui est décrit dans le cahier des charges. Par expérience, on s'est aperçu que l'implantation posait souvent des difficultés. Par exemple, après une mauvaise implantation, on peut obtenir des programmes qui s'arrêtent en raison d'erreurs, des calculs erronés ou bien encore des effets de bords dus à des dépassements de certaines limites. D'où l'apparition d'outils offrant la possibilité d'exécuter pas à pas un programme, ce qui permet d'analyser son comportement et de détecter d'éventuels dysfonctionnements. Une fois qu'on a obtenu un programme qui s'exécute, se pose la question de savoir s'il s'exécute correctement. Pour répondre à cette question, on peut entreprendre une série de tests qui doivent révéler la présence d'erreurs ou de problèmes survenant à l'exécution. Finalement, il est primordial de savoir s'il s'exécutera toujours correctement, quels que soient les paramètres qu'il recevra ou les données qu'il traitera. Les méthodes formelles sont apparues afin de répondre à cette attente.
Le but des méthodes formelles est d'offrir un moyen pour raisonner sur les programmes. Suivant la manière de modéliser un système et les propriétés qu'on souhaite vérifier, certaines méthodes formelles sont plus adaptées que d'autres.

Concernant les programmes parallèles, l'explosion des exécutions possibles et leur non reproductibilité rend la validation par le test très difficile. C'est pourquoi les méthodes formelles nous intéressent particulièrement pour prouver le comportement d'un programme parallèle.

2.1.2 Quelles sont les différentes catégories?

De nombreux documents détaillent les différences et les enjeux liés aux méthodes formelles, par exemple [Clarke et Wing, 96]. Certaines méthodes formelles sont fondées sur les ensembles, on peut citer VDM [Jones, 1986] et Z [Spivey, 1987] comme précurseurs et plus récemment la méthode B [Abrial, 1996]. D'autres méthodes utilisent la notion d'état et de systèmes de transitions pour représenter une spécification. TLA [Lamport, 1994], UNITY [Chandy et Misra, 1988] en sont deux exemples, B a également cette caractéristique. Certaines personnes ont préféré utiliser des types, on peut citer PVS [Owre *et al.*, 1998] et Isabelle [Paulson, 1994] par exemple. Certains modèles utilisent les types abstraits algébriques, LOTOS [Bolognesi et Brinksma, 1987] en fait partie. LOTOS est également fondé sur les algèbres de processus comme CCS [Milner, 1980] et CSP [Hoare, 1985]. La tendance actuelle pour ces modèles est d'intégrer la notion d'état. Il faut également citer tous les travaux sur le *model checking*, [Clarke et Emerson, 1981; Clarke *et al.*, 1994] en sont des exemples. L'objectif est de vérifier que des propriétés sont correctes sur un modèle fini d'un système. L'intérêt est que la vérification est automatique. L'inconvénient est qu'on est rapidement confronté à une explosion du nombre d'états à parcourir.
On peut également distinguer les méthodes qui permettent d'effectuer une correction a posteriori, des méthodes qui permettent d'assurer une correction a priori. Dans le premier cas, on prouve que le programme est correct une fois qu'il est achevé, alors que dans le second cas, on effectue la preuve dès la conception en utilisant le raffinement. Le principe du raffinement est d'appliquer des transformations sur un programme ou une spécification en prenant soin par l'intermédiaire de preuves que celles-ci conservent les propriétés du programme ou de la spécification.
Maintenant nous allons détailler les méthodes qui nous paraissent intéressante pour notre étude.

2.2 TLA

TLA (Temporal Logic of Actions) est une logique définie par Lamport [Lamport, 1994] en 1991, pour spécifier et raisonner sur les systèmes concurrents. Les systèmes et leurs propriétés sont représentés dans la même logique. Ainsi, les propriétés telles qu'un système satisfait sa spécification, et qu'un système en raffine un autre, s'expriment par des implications logiques. Le langage de spécification comporte une particularité intéressante car il n'est pas typé. Il se différencie donc de la plupart des autres langages de spécification par ce fait. Lamport a écrit de nombreux articles à propos de TLA, la manière de spécifier avec cette logique et avec le temps, il l'a enrichi pour déboucher sur le langage de spécification TLA$^+$ [Lamport, 1997; Lamport, 1998]. TLA$^+$ est fondé sur la théorie des ensembles de Zermelo-Fraenkel avec l'axiome du choix. La notion de module permet de structurer les spécifications. Dans la suite nous nous intéresserons à TLA et pas spécialement à TLA$^+$.

2.2.1 Définitions pour TLA

- Une *variable*, puisqu'elle n'a pas de type, peut prendre n'importe quelle valeur, aussi bien une valeur booléenne, entière, réelle ou bien une chaîne de caractères... Une variable *flexible* est une variable dont la valeur peut changer, à l'opposé d'une variable *rigide* qui possède

une valeur qui ne change pas. Dans ce cas, on garde le terme variable car on ne connaît pas la valeur de cette variable.

- Un *état* est défini comme une valuation des variables. Donc un état s est une fonction surjective de l'ensemble des variables vers l'ensemble des valeurs possibles.
- Une *trace* est définie comme une suite infinie d'états.
- On définit un *prédicat* comme une expression booléenne construite à partir des variables et des constantes. Par exemple, $x-y = 23$, ou $chat \in ANIMAUX$ sont des prédicats.
- Une *action* est une expression booléenne composée de variables, variables primées et de symboles de constantes. Par exemple, $x' = x+2$, ou $E' = E \cap F$, sont des actions. Une action représente une relation entre un état courant et son successeur. Les variables non primées font référence à l'état courant et les variables primées font référence à l'état suivant. Donc $x' = x+2$ signifie que la nouvelle valeur de x est égale à la précédente plus 2.
- Un prédicat peut être primé, dans ce cas, toutes les variables qui le composent sont primées. Par exemple, si on a le prédicat $I \triangleq x > 0$ alors I' signifie $x' > 0$.
- Le *bégaiement* (stuttering) désigne une suite d'états où certaines variables ne changent pas de valeur. Il est utile pour le raffinement de cacher certaines variables. Dans TLA, deux traces qui ne diffèrent que par le bégaiement satisfont les mêmes formules. On exprime le bégaiement sur une action de la manière suivante : $[A]_f \triangleq A \vee (f' = f)$, où f représente un tuple de variables. Cette formule signifie que l'on a soit l'action A, soit toutes les variables représentées par f à l'état suivant sont égales à celles de l'état courant, c'est-à-dire que l'on ne fait rien, c'est le bégaiement.
- À l'inverse du bégaiement, on peut avoir besoin d'exprimer qu'une action sera exécutée et que certaines variables seront différentes après l'exécution de cette action. On exprime cette propriété avec la règle suivante : $\langle A \rangle_f \triangleq A \wedge (f' \neq f)$.
- Pour toute action $\mathcal{A}$, le prédicat *Enabled* $\mathcal{A}$ est vrai pour un état, si et seulement si, l'action $\mathcal{A}$ peut être exécutée. Plus précisément, si $x_1, \ldots, x_n$ sont les variables libres qui apparaissent primées dans $\mathcal{A}$, alors *Enabled* $\mathcal{A}$ est égal à $\exists\, x'_1, \ldots, x'_n : \mathcal{A}$,

2.2.2 Logique temporelle pour TLA

On construit une formule temporelle à partir des prédicats, des opérateurs booléens et de l'opérateur $\Box$ (toujours ou always). Par exemple avec les formules non temporelles A et B, on peut construire la formule temporelle $\Box(A \wedge B)$, celle-ci signifie que $A \wedge B$ est *toujours* vraie.
L'opérateur dual à $\Box$ est $\Diamond$, cet opérateur signifie *un jour*. Il est défini de la manière suivante : $\Diamond F \triangleq \neg\Box\neg F$.
Avec ces deux opérateurs, on construit par composition, les opérateurs $\Box\Diamond$ et $\Diamond\Box$. La formule $\Box\Diamond\, F$ signifie que F est *infiniment souvent* vraie. La formule $\Diamond\Box\, F$ signifie quant à elle que F est *un jour toujours* vraie.
En logique temporelle, on utilise également l'opérateur *leads to* représenté par $\rightsquigarrow$. Cet opérateur est défini par $A \rightsquigarrow B$ est équivalent à $\Box(A \Rightarrow \Diamond B)$. Il permet d'exprimer la vivacité du système dans le sens où la formule $A \rightsquigarrow B$ signifie que la formule A conduit nécessairement (sans préciser quand) à B.
TLA possède deux équités, pour spécifier que si une opération ou une action doit être exécutée et qu'elle reste exécutable alors un jour elle sera exécutée. La première est l'équité forte qui dit que l'action sera infiniment souvent exécutée si elle est infiniment souvent exécutable. La seconde est l'équité faible qui spécifie que l'action sera infiniment souvent exécutée si un jour elle est toujours exécutable.

Voici la définition de ces équités dans TLA :
$\mathrm{SF}_f(A) \triangleq \Box\Diamond \mathit{Enabled}\ \langle A\rangle_f \Rightarrow \Box\Diamond \langle A\rangle_f$
$\mathrm{WF}_f(A) \triangleq \Diamond\Box \mathit{Enabled}\ \langle A\rangle_f \Rightarrow \Box\Diamond \langle A\rangle_f$

Pour représenter un système, un programme ou un algorithme, on spécifie quels sont les comportements autorisés par celui-ci. En exprimant la spécification sous la forme d'une formule TLA, un système est spécifié par la formule correspondant à un ensemble de comportements autorisés. La forme générale canonique d'une spécification TLA est la suivante :
$\mathit{Init} \wedge \Box[N]_f \wedge F$
Dans cette formule :

- *Init* est un prédicat qui spécifie les valeurs initiales des variables.
- N est l'ensemble des actions qui donnent le prochain état du système. Les actions sont les opérations atomiques du programme.
- f est le tuple de toutes les variables flexibles.
- F est la conjonction de toutes les formules d'équité : $\mathrm{WF}_f(A)$ et $\mathrm{SF}_f(A)$, où A est une action qui représente un sous-ensemble des opérations du programme.

Cette formule agit sur l'ensemble des traces possibles. *Init* réduit l'ensemble des traces possibles puisqu'il donne l'état initial. L'ensemble N des actions exécutables réduit encore l'ensemble des traces car seulement certaines actions sont exécutables. Finalement, les conditions d'équités contraignent les traces afin que les choses bonnes arrivent un jour.

2.2.3 Un petit exemple

Cet exemple trie les éléments d'un tableau. Dans cette spécification, l'action *Init* initialise la valeur n et définit le tableau A. La notation $[1..n \rightarrow \mathit{Real}]$ signifie que nous avons une fonction qui va de l'ensemble des nombres entiers de 1 à n dans l'ensemble des réels. L'action *Exchange* a pour but d'échanger deux éléments consécutifs si le premier est supérieur au second. Dans cette action, on utilise la notation préfixée utilisée par Lamport pour les conjonctions et les disjonctions. Le mot clé EXCEPT permet de modifier la valeur d'une fonction en un seul point. Donc pour cette action $A' = [A \text{ EXCEPT } ![x] = A[x+1], ![x+1] = A[x]]$ signifie que la variable A' est égale à la valeur de A dans laquelle les valeurs de $A[x]$ et de $A[x+1]$ ont été permutées. On met l'équité faible sur toutes les actions *Exchange* par l'intermédiaire de *Fair*, afin d'assurer que tous les échanges puissent se produire.

MODULE *SORT*

CONSTANT n
VARIABLES A
$\mathit{Vars} \triangleq \langle A\rangle$

Initialisation des variables
$\mathit{Init} \triangleq \wedge\ n > 1$
$\qquad\qquad \wedge\ A \in [1..n \rightarrow \mathit{Real}]$

Echange de deux valeurs
$\mathit{Exchange}(x) \triangleq \wedge\ A[x] > A[x+1]$
$\qquad\qquad\qquad\qquad \wedge\ A' = [A \text{ EXCEPT } ![x] = A[x+1], ![x+1] = A[x]]$

$Next \triangleq \exists\, m \in 1..(n-1) : Exchange(m)$

$Fair \triangleq \forall\, m \in 1..(n-1) : \mathrm{WF}_{\langle Vars \rangle}(Exchange(m))$

Propriété pour vérifier que les nombres sont triés

$Sorted \triangleq \forall\, m \in 1..(n-1) : A[m] \leq A[m+1]$

$Spec \triangleq (Init \wedge \Box[Next]_{\langle Vars \rangle} \wedge Fair)$

Théorème

$Tri\_Correct \triangleq Spec \Rightarrow Init \rightsquigarrow Sorted$

2.2.4 Preuve avec TLA

Nous n'allons pas détailler comment faire des preuves avec TLA, nous allons seulement donner l'idée générale.
Une spécification a l'allure que nous avons donnée précédemment. Maintenant nous allons décrire schématiquement ce qu'il faut faire pour prouver une propriété avec cette spécification. Chaque fois que nous utilisons une règle, nous décrivons la catégorie à laquelle elle appartient.
Tout d'abord quels types de propriétés peut on prouver?

- Les propriétés d'invariance, du style : $Spec \Rightarrow \Box P$
- Les propriétés de vivacité, du style : $Spec \Rightarrow P \rightsquigarrow Q$
- Les propriétés de raffinement sont de la forme $\phi \Rightarrow \psi$, où ϕ et ψ sont des spécifications. Dans ce cas, on dit que ϕ est un raffinement de ψ, car il réalise la même chose que ψ mais de manière plus précise. ψ est plus abstrait que ϕ.

Maintenant, examinons pour chaque propriété comment la prouver, sachant que I, P et Q sont des prédicats, A et N sont des actions, H est une formule TLA :

- Les propriétés d'invariance sont prouvées généralement avec la règle :

$$\frac{I \wedge [N]_f \Rightarrow I'}{I \wedge \Box[N]_f \Rightarrow \Box I} \qquad \text{(INV1)}$$

  Cette règle transforme une formule temporelle en une formule non temporelle qui est prouvée en utilisant la logique classique.
- Les propriétés de vivacité sont prouvées à l'aide de trois règles :
  - La *lattice rule* utilise l'opérateur $\prec$ qui définit un ordre bien fondé. Cette règle permet de prouver que si un état satisfait une propriété $H(x)$ pour tout x, suivi d'un état G ou d'un état $H(y)$ avec y tel que $y \prec x$, alors tout état qui satisfait $H(z)$ conduit à un état qui satisfait G.

$$(\forall x : H \rightsquigarrow (G \vee \exists y :\ y \prec x \wedge H[y/x])) \Rightarrow ((\exists\, x :\ H) \rightsquigarrow G) \qquad \text{(LATTICE)}$$

- La règle *WF1* élimine l'opérateur *leads to* grâce à l'équité faible dans une formule :

$$\frac{\begin{array}{l} P \wedge [N]_v \Rightarrow P' \vee Q' \\ P \wedge \langle N \wedge A \rangle_v \Rightarrow Q' \\ P \Rightarrow \mathit{Enabled}\ \langle A \rangle_v \end{array}}{\Box[N]_f \wedge \mathrm{WF}_f(A) \Rightarrow (P \rightsquigarrow Q)} \qquad \text{(WF1)}$$

- La règle *SF1* élimine l'opérateur *leads to* grâce à l'équité forte dans une formule :

$$\frac{\begin{array}{l} P \wedge [N]_v \Rightarrow P' \vee Q' \\ P \wedge \langle N \wedge A \rangle_v \Rightarrow Q' \\ \Box P \wedge \Box[N]_f \wedge \Box F \Rightarrow \Diamond \mathit{Enabled}\ \langle A \rangle_f \end{array}}{\Box[N]_f \wedge \mathrm{SF}_f(A) \wedge \Box F \Rightarrow (P \rightsquigarrow Q)} \qquad \text{(SF1)}$$

  Avec ces deux règles d'équité on cherche à montrer qu'une propriété P conduit à une propriété Q. À partir de ces règles d'élimination de l'opérateur *leads to*, on obtient 3 formules dont les deux premières ne contiennent plus d'opérateurs temporels. Ces deux premières formules sont identiques pour les deux équités. La première règle signifie qu'à partir d'une propriété P et pour toute action de l'ensemble des actions, si on accepte le bégaiement alors on obtient la propriété P ou la propriété Q à l'état suivant, c'est-à-dire qu'on conserve au moins la propriété P qu'on avait en hypothèse. La seconde règle signifie qu'à partir de la propriété P et en exécutant sans bégaiement une action particulière A qui appartient à l'ensemble des actions, alors on obtient la propriété Q à l'état suivant. La troisième règle de $WF1$ consiste à prouver que P rend activable l'action A qui nous permet d'atteindre notre but. La troisième règle de $SF1$ demande de prouver que $\Box P$, toutes les actions, et une condition $\Box F$ qui, dans la pratique, est une action qui fait « avancer » le système, rendent un jour l'action A activable. En général, la troisième règle de $WF1$ est plus facile à prouver que celle de $SF1$.

- Les vérifications de raffinement de la forme $\phi \Rightarrow \psi$ sont prouvées à l'aide de trois conditions :
  - La première consiste à prouver que les conditions initiales de ϕ impliquent celles de ψ, c'est-à-dire que $\mathit{Init}_\phi \Rightarrow \mathit{Init}_\psi$.
  - La seconde nécessite de prouver que les actions de ϕ impliquent les actions de ψ, autrement dit que $\Box[N_\phi]_{f_\phi} \Rightarrow \Box[N_\psi]_{f_\psi}$
  - Finalement la troisième demande que ϕ implique les conditions d'équité de ψ, soit $\phi => \bigwedge_i \mathrm{SF}_{f_i}(A_{\psi_i}) \bigwedge_j \mathrm{WF}_{f_j}(A_{\psi_j})$. Pour les formules de cette forme on utilise deux règles $WF2$ et $SF2$ qui permettent de les prouver. Celles-ci se trouvent dans [Lamport, 1994].

Il existe une autre forme de raffinement qui permet de prouver qu'une spécification $Spec_1$ simule une autre spécification $Spec_2$. Cette technique s'appelle fonctions de raffinement (*refinement mappings*). Elle consiste à simuler les variables y de $Spec_1$ par une fonction $\overline{y}$ des variables x de $Spec_2$. Cette simulation est validée par la preuve de $Spec_2 \Rightarrow Spec_1[\overline{y}/y]$. $Spec_1[\overline{y}/y]$ représente $Spec_1$ où $\overline{y}$ est substitué à y. Cette technique permet de définir globalement un raffinement entre deux spécifications. Pour prouver une fonction de raffinement, il est parfois nécessaire d'introduire des variables d'histoire ou de bégaiement. On peut trouver des exemples d'emploi de variables auxiliaires dans les preuves de [Ladkin *et al.*, 1996].

2.3 UNITY

UNITY a été défini par Chandy et Misra [Chandy et Misra, 1988]. Il est le fruit de la conviction des auteurs que les programmes peuvent (et doivent) être conçus selon une méthode qui est indépendante de l'architecture et du domaine de l'application. UNITY est une théorie simple et universelle dans laquelle on représente de manière uniforme les différents types d'applications : synchrones, asynchrones, à mémoire partagée ou à base d'échange de messages, etc. Le formalisme d'UNITY offre la possibilité de représenter un programme sous différents angles, du plus abstrait (la spécification) au plus concret (l'implantation) dans la même théorie. Ainsi, il permet de concevoir des programmes par raffinement. Le formalisme d'UNITY est composé d'un langage de programmation, inspiré du langage des commandes gardées de Dijkstra [Dijkstra, 1976], d'un langage de spécification fondé sur une logique temporelle linéaire et une théorie permettant d'effectuer la preuve, sur les programmes, des propriétés exprimées dans le langage de spécification. Le modèle opérationnel d'UNITY est fondé sur un système de transitions dans lequel les transitions sont représentées par les affectations.

2.3.1 Structure d'un programme UNITY

Afin de comprendre pourquoi la notation UNITY est simple, nous allons la détailler assez rapidement. La structure initiale d'un programme UNITY est composée :

- d'une section *declaration* utilisée pour définir les variables,
- d'une section *always* dans laquelle nous définissons certaines variables du programme comme des fonctions d'autres variables,
- d'une section *initially* pour initialiser les variables et
- d'une section *assign* contenant l'ensemble de toutes les affectations considérées comme des actions atomiques.

La section *assign* joue le rôle principal d'un programme UNITY car tous les calculs sont réalisés avec elle. Une affectation peut être simple ou multiple, par exemple :

x,y,z := 0,1,2

correspond à :

$x := 0 \ \| \ y := 1 \ \| \ z := 2$, où l'opérateur || exprime le parallélisme entre les affectations.

L'opérateur [] exprime le non-déterminisme. Donc, par exemple, si on a les instructions suivantes : $x := 0 \ [] \ y := 1 \ [] \ z := 2$, une des trois instructions sera exécutée mais on ne sait pas laquelle. Il y a quand même une hypothèse d'équité faible dans le choix.
Les quantifications sont autorisées et elles sont plus simples que les énumérations, par exemple :

$\langle \| \ i : 0 \leq i \leq N :: A[i] := B[i] \rangle$

correspond à :

$A[0] := B[0] \ \| \ .. \ \| \ A[i] := B[i] \ \| \ .. \ \| \ A[N] := B[N]$

Les affectations peuvent être gardées, donc $a := b \ \ if \ \ a > b$ exécutera l'affectation $a := b$

seulement si l'expression $a > b$ est vraie. Les gardes offrent la possibilité d'exprimer le non-déterminisme dans un programme. C'est un avantage si on souhaite raisonner sur un programme abstrait mais, au contraire, elles introduisent des complications qui vont à l'encontre des performances lorsque l'on s'intéresse au code.

2.3.2 Un petit exemple

Afin de constater la simplicité de cette notation, nous donnons un petit exemple, qui effectue simplement un tri sur un tableau de manière séquentielle, tiré de [Chandy et Misra, 1988].

```
program Sort
assign
⟨[] i : 0<=i<N :: A[i],A[i+1] := A[i+1],A[i] if A[i]>A[i+1]⟩
end
```

2.3.3 Preuves avec UNITY

La logique d'UNITY est basée sur des assertions de la forme $\{p\}\ s\ \{q\}$ qui dénote que l'exécution d'une instruction s dans n'importe quel état qui satisfait le prédicat p résulte par un état qui satisfait le prédicat q, si l'instruction s se termine. Un problème de consistance a été trouvé par Sanders [Sanders, 1991] qui oblige à définir les règles initiales de manière plus compliquée. Le problème vient du fait que dans la définition initiale des règles d'UNITY, on peut considérer des états inaccessibles d'un programme. L'explication détaillée est faite dans l'article de Sanders ou dans la thèse de Charpentier [Charpentier, 1997]. La solution proposée pour contourner ce problème consiste à faire la distinction entre les états accessibles et les états inaccessibles d'un programme. Les états inaccessibles sont évités grâce au « plus fort invariant ». Pour un programme donné F, les contraintes d'invariance (être vrai dans les états initiaux et être maintenu pour toutes les transitions) constituent une équation en terme de prédicat décrite par une fonction monotone (au sens de l'implication). On peut alors montrer, comme l'explique Charpentier, que cette équation admet une solution maximale baptisée *plus fort invariant* du programme F. On la note $F.SI$ (pour *strongest invariant*). Ce prédicat peut être vu comme la caractérisation des états accessibles d'un programme : un état est accessible au cours d'une exécution du programme F si et seulement s'il satisfait le prédicat $F.SI$.
Voici les quatre règles principales dans UNITY pour prouver des propriétés de sûreté et d'invariance.

- La propriété *Unless* est définie de la manière suivante :

$$p \textit{ unless } q \equiv \langle \forall s : s \in F :: \{p \wedge \neg q \wedge F.SI\}\ s\ \{p \vee q\}\rangle$$

  Cette propriété signifie que si p est vrai à un certain moment et que q ne l'est pas, alors dans l'étape suivante (après l'exécution de s), p reste vrai ou q devient vrai.
- La propriété *Ensures* est définie par :

$$p \textit{ ensures } q \equiv (p \textit{ unless } q \wedge \langle \exists\, s : s \in F :: \{p \wedge \neg q \wedge F.SI\}\ s\ \{q\}\rangle)$$

- Le *leadsto*, représenté par l'opérateur $\mapsto$ s'obtient à partir de l'une des trois règles :

$$\frac{P \textit{ ensures } Q}{P \mapsto Q}$$

$$\frac{P \mapsto R, R \mapsto Q}{P \mapsto Q}$$

$$\frac{\langle\, \forall m : m \in W :: P(m) \mapsto Q \,\rangle}{\langle\, \exists m : m \in W :: P(m) \,\rangle \mapsto Q}$$

- L'axiome de substitution est alors :
  Si $x = y$ est un invariant de F, alors x peut remplacer y dans toutes les propriétés de F. Cet axiome s'inspire de la règle de Leibniz en logique qui exprime que l'on peut substituer dans une formule logique une partie par une formule équivalente sans changer la valeur de vérité de la formule.

2.4 Présentation du prouveur PVS

PVS (Prototype Verification System) est un environnement de spécification et de vérification [Crow *et al.*, 1995; Owre *et al.*, 1998]. Il fournit des outils pour créer et analyser les spécifications formelles et prouver interactivement des théorèmes. Il a été utilisé dans des domaines très variés tels que l'avionique [Dutertre et Stavridou, 1997], la vérification de systèmes réactifs [Kellomäki, 1997; Saïdi, 1997] et le développement de programme [Hooman, 1997].

Le langage de spécification

Le langage de spécification de PVS est basé sur la logique typée d'ordre supérieur et il possède un système de type riche qui inclut les constructeurs, les types dépendants, les types de données abstraits et les mécanismes de sous-typage. Ces caractéristiques nous permettent d'exprimer des spécifications puissantes, et la vérification de type nous aide à éviter de nombreuses erreurs sémantiques. Les spécifications trop grandes sont découpées en plusieurs théories afin de former une hiérarchie de théories réutilisables. Toutes les fonctions dans PVS sont totales, certains peuvent y voir un inconvénient, mais la notion de *predicate subtype*, un type qui garde uniquement les éléments d'un autre type qui satisfont un prédicat donné, remédie à celui-ci. Avec les *predicate subtype*, on définit le type de l'opérateur de division, par exemple, avec un numérateur arbitraire et un dénominateur différent de zéro. Le domaine de l'opération *pop* sur une pile est de manière similaire restreint à une pile non vide. Nous donnons l'exemple de la pile repris de [Owre *et al.*, 1998].

```
stacks [t: TYPE+] : THEORY
  BEGIN

  stack : TYPE+
  s : VAR stack
  empty : stack
  nonemptystack?(s) : bool = s /= empty

  push : [t, stack -> (nonemptystack?)]
  pop : [(nonemptystack?) -> stack]
  top : [(nonemptystack?) -> t]

  x, y : VAR t
```

```
  push_top_pop : AXIOM
    nonemptystack?(s) IMPLIES push(top(s), pop(s)) = s

  pop_push : AXIOM pop(push(x, s)) = s

  top_push : AXIOM top(push(x, s)) = s

  pop2push2 : THEOREM pop(pop(push(s, push(y, s)))) = s

END stacks
```

Cette spécification n'est pas forcément recommandée pour spécifier une pile car nous introduisons des axiomes, et si l'un d'eux est faux, notre spécification n'est pas consistante. Néanmoins, sa simplicité permet d'introduire les notions de base du langage de spécification de PVS. La première ligne de cette spécification définit une théorie que nous appelons `stacks`. Cette théorie est paramétrée par un type `t` (le paramètre formel de `stacks`). Le mot clé `TYPE+` indique que `t` est un type non vide. Ensuite on définit le type non interprété (et non vide) `stack`. La variable `s` et la constante `empty` sont définies de type `stack`. Le prédicat `nonemptystack?` est ensuite déclaré sur des éléments de type `stack` ; il est vrai pour un élément donné `stack`, si et seulement si, cet élément n'est pas égal à `empty`. Ensuite on déclare les fonctions `push`, `pop` et `top`. Notons que le prédicat `nonemptystack?` est utilisé comme un type dans la signature de ces fonctions.
Les variables `x` et `y` sont ensuite déclarées, suivies des axiomes classiques concernant `push`, `pop` et `top`, qui font de `push` un constructeur et de `pop` et de `top` des observateurs de `stack`. Finalement le théorème `pop2push2`, peut être prouvé facilement avec deux applications de l'axiome `pop_push`. Ce théorème simple possède une facette cachée que l'on découvre durant la phase de vérification de type. Nous avons spécifié que `pop` attend un élément de type `(nonemptystack?)` et retourne une valeur de type `stack`. Cela fonctionne parfaitement pour le `pop` qui se trouve à l'intérieur parce qu'il reçoit du `push` un élément de type `(nonemptystack?)` ; mais pour le `pop` de l'extérieur, son type ne peut pas être vérifié de manière syntaxique. Dans ce cas, le système génère une condition de correction de type (Type-Correctness Condition). Dans notre cas, elle a la forme suivante :

```
pop2push2_TCC1: OBLIGATION
      (FORALL (s: stack, x: t, y: t):
        nonemptystack?(pop(push(x, push(y, s)))))
```

Nous donnons une liste non exhaustive des possibilités du langage de PVS. On peut déclarer des types, des types abstraits (ou non interprétés), des sous-types abstraits, des types interprétés, des types énumérés, des constantes, des variables. On peut utiliser des définitions récursives, inductives. On peut déclarer des fonctions, des tuples, des enregistrements. Concernant les expressions, on peut exprimer les expressions booléennes, les expressions `IF-THEN-ELSE`, les expressions numériques, les applications, les quantifications et les tables. Ces dernières sont utilisées, par exemple, pour spécifier le comportement d'une fonction suivant certaines valeurs (en voici un exemple : pour $x < 0$ on renvoie -1, pour $x = 0$ on renvoie 0 et pour $x > 0$ on renvoie 1). Les théories permettent de décomposer une spécification et de réutiliser certaines parties à l'aide des clauses `IMPORTING` et `EXPORTING`. Finalement on peut utiliser les types abstraits de données algébriques et ainsi la spécification de la pile est plus naturelle.
Le prouveur de PVS possède un ensemble de commandes qui peuvent être utilisées pour simplifier le but courant automatiquement. Les théorèmes simples sont prouvés de manière automatique

grâce à des tactiques définies dans PVS, mais lorsque les théorèmes sont plus compliqués, l'utilisateur doit guider la preuve à l'aide du prouveur interactif. Le prouveur de théorème de PVS est fondé sur le calcul des séquents. Ainsi, prouver un théorème consiste à prouver qu'un séquent équivalent est vrai. Pendant la preuve, le séquent courant — le but — subit des transformations qui le rendent vrai ou génèrent des sous-buts à prouver. Les commandes de PVS sont paramétrables et ainsi, l'utilisateur a de nombreux moyens de mener à bout sa preuve.
Chaque transformation de séquent est réalisée avec une commande prédéfinie de PVS. Le système contient plus d'une centaine de commandes, plus ou moins complexes. En général, on essaie d'appliquer des commandes de haut niveau, telles que l'induction, la réécriture, l'instanciation avec des heuristiques... Avec de l'expérience, on connaît le fonctionnement de chaque commande. Après l'application d'une suite de commandes, on termine la preuve en utilisant des procédures de décision et de simplification puissantes pour l'arithmétique linéaire et l'égalité. Toutes ces commandes peuvent être combinées pour former des stratégies de preuve, permettant ainsi d'appliquer plusieurs règles de preuve en un seul pas.
Globalement faire une preuve avec un prouveur nécessite plus de temps qu'une preuve effectuée à la main, mais on obtient la certitude qu'il n'y a pas d'erreur. Et plus la spécification est longue plus la probabilité de faire des erreurs dans une preuve à la main est importante.
Voici la preuve du théorème `pop2push2` pour comprendre comment on utilise PVS. Au départ on a un séquent qui contient le but à prouver. Dans un séquent, on différencie les formules qui sont en hypothèses (elles ont des numéros négatifs), des formules qui appartiennent au but (elles ont des numéros positifs).

```
pop2push2 :

  |-------
1    (FORALL (s: stack, x: t, y: t): pop(pop(push(x, push(y, s)))) = s)
```

En appliquant la commande `SKOSIMP *`, on skolemise et on monte en hypothèse toutes les formules tant que c'est possible.

```
Rerunning step: (SKOSIMP*)
Repeatedly Skolemizing and flattening,
this simplifies to:
pop2push2 :

  |-------
1    pop(pop(push(x!1, push(y!1, s!1)))) = s!1
```

Les variables `s`, `x` et `y` ont été skolemisées par les variables `s!1`, `x!1` et `y!1`. Ensuite on applique le lemme `pop_push`. Pour cela, on utilise la commande `LEMMA`.

```
Rerunning step: (LEMMA "pop_push")
Applying pop_push
this simplifies to:
pop2push2 :

-1    (FORALL (s: stack, x: t): pop(push(x, s)) = s)
  |-------
[1]    pop(pop(push(x!1, push(y!1, s!1)))) = s!1
```

Donc ce lemme est monté en hypothèse. Maintenant on instancie les variables quantifiées de ce lemme. Cette instanciation est réalisée par la commande `INST` à laquelle on précise la formule

que l'on instancie : -1. Ensuite on indique à PVS que push(y!1, s!1) doit instancier s et que x!1 instancie x.

```
Rerunning step: (INST -1 "push(y!1, s!1)" "x!1")
Instantiating the top quantifier in -1 with the terms:
 push(y!1, s!1), x!1,
this simplifies to:
pop2push2 :

-1    pop(push(x!1, push(y!1, s!1))) = push(y!1, s!1)
  |-------
[1]    pop(pop(push(x!1, push(y!1, s!1)))) = s!1
```

Puis on indique à PVS que l'on souhaite qu'il remplace chaque fois que c'est possible la partie gauche de la formule -1 par la partie droite avec la commande REPLACE.

```
Rerunning step: (REPLACE -1)
Replacing using formula -1,
this simplifies to:
pop2push2 :

[-1]    pop(push(x!1, push(y!1, s!1))) = push(y!1, s!1)
  |-------
1    pop(push(y!1, s!1)) = s!1
```

Finalement, on utilise le lemme pop_push que l'on instancie tout simplement comme suit.

```
Rerunning step: (LEMMA "pop_push")
Applying pop_push
this simplifies to:
pop2push2 :

-1    (FORALL (s: stack, x: t): pop(push(x, s)) = s)
[-2]    pop(push(x!1, push(y!1, s!1))) = push(y!1, s!1)
  |-------
[1]    pop(push(y!1, s!1)) = s!1

Rerunning step: (INST -1 "s!1" "y!1")
Instantiating the top quantifier in -1 with the terms:
 s!1, y!1,
Q.E.D.
```

2.5 Conclusion

Ce chapitre a présenté très brièvement les méthodes formelles. TLA et UNITY sont des méthodologies, fondées sur la logique temporelle, qui ont toutes deux pour but d'offrir un moyen de développer des programmes concurrents en utilisant des raffinements. Nous avons choisi de présenter particulièrement ces deux méthodes formelles car elles ont présenté une grande avancée en matière de spécification et de cadre pour effectuer des preuves de propriétés pour les langages parallèles. Bien que nous ne l'ayons pas mentionné dans la présentation d'UNITY, UNITY se distingue par rapport à TLA car les auteurs se sont également intéressés à l'exécution des pro-

grammes UNITY. Toutefois il existe peu de travail à ce sujet c'est pourquoi nous n'avons rien présenté.
Nous n'avons pas réalisé de longues preuves avec TLA et UNITY parce qu'il n'existe pas d'outils adaptés afin de mécaniser les preuves. Mais à chaque fois que nous avons utilisé PVS afin de réaliser des preuves, nous avons gardé en tête les principes de ces deux modèles. À divers moments pendant notre travail nous avons tenté de voir s'il était possible de mécaniser des preuves de TLA et d'UNITY dans PVS, mais ce n'est pas réalisable facilement[5]. Actuellement, il est possible de prouver des propriétés écrites dans ces deux langages avec Isabelle [Paulson, 1999; Merz, 1997]. Pour TLA, l'auteur du mode TLA d'Isabelle reconnaît lui-même qu'il est plus difficile de prouver une spécification avec son outil que de la prouver à la main. Pour UNITY, l'outil est très récent et nous n'avons pas eu le temps de l'utiliser. L'avantage d'Isabelle par rapport aux autres outils de preuves réside dans la facilité qu'il procure pour encoder un autre système de preuve (comme TLA et UNITY). Le désavantage d'Isabelle provient du manque d'automatisation dans la recherche de preuves par rapport aux autres outils de preuves, notamment PVS.
DISCO [Kurki-Suonio, 1998] est inspiré du travail réalisé par Lamport sur TLA. Il a pour but de spécifier formellement des systèmes réactifs. Le langage de spécification est assez proche de TLA mais également d'UNITY. L'intérêt de DISCO réside dans les outils qui ont été développés autour de celui-ci. Actuellement il existe un outil pour animer les exécutions de spécifications, un outil pour visualiser les scénarios. Une ébauche de mécanisation avec PVS a également été réalisée.
Notre choix pour prouver qu'une parallélisation est correcte s'est porté vers PVS qui utilise la logique classique plutôt que sur TLA ou UNITY qui utilise la logique temporelle car nous voulons prouver que les résultats du programme paralléle sont identiques à ceux du programme séquentiel. La logique temporelle est bien adaptée pour montrer qu'un comportement finira par se produire et donc qu'il n'y aura pas de bloquage. En utilisant OpenMP, nous sommes sûr qu'il n'y aura pas de bloquage car le compilateur s'occupe de tout. De manière générale, en réalisant une abstraction sur notre programme parallèle, nous modélisons les communications ou nous en faisons une abstraction. Donc l'apport de la logique temporelle ne nous est pas indispensable, d'où notre choix pour un outil permettant de prouver de la logique classique, d'autant plus que nous recherchons un outil qui aura la possibilité de faire le maximum de travail tout seul. PVS est pour l'instant l'outil le plus satisfaisant en respectant ces critères.

5. En fait pour TLA, le problème vient du fait qu'il faut pouvoir différencier la logique temporelle de la logique classique et PVS ne travaille qu'avec la logique classique. Pour UNITY, le problème de consistance fait qu'on est obligé de calculer le plus fort invariant, ce qui complique la mécanisation.

3

Méthodes formelles et parallélisme : l'existant

Ce chapitre décrit des travaux qui ont utilisé les méthodes formelles pour développer des programmes parallèles. Nous présentons les méthodologies d'Owicki-Gries [Owicki et Gries, 1976], de Massingill [Massingill, 1998] et de Sere [Sere, 1990] pour développer des programmes parallèles. Ces deux dernières méthodologies ont été utilisées pour développer des programmes réels. Ce chapitre présente également les travaux de Foster et Darlington concernant le parallélisme. Ainsi nous pouvons situer notre travail par rapport au contexte existant.

3.1 Une technique de preuve axiomatique pour les programmes parallèles

Owicki et Gries ont défini une technique pour prouver des propriétés d'algorithmes parallèles [Owicki et Gries, 1976]. Elle étend l'axiomatique de Hoare [Hoare, 1969]. Cette technique permet de prouver la correction partielle, l'absence de blocage et la terminaison d'algorithmes parallèles.

3.1.1 Le langage, les axiomes et les propriétés

Le langage utilisé pour décrire les programmes est celui de Hoare pour les parties séquentielles. On utilise la notion de triplet de la forme $\{P\}\ S\ \{Q\}$. Avec une telle notation, P est une propriété logique qui est vraie avant l'exécution de l'instuction S et Q est une propriété qui est vraie après l'exécution de S. P est une pré-condition alors que Q est une post-condition. Les règles pour prouver des propriétés sur les programmes séquentiels sont les suivantes :

- instruction inopérante

$$\{P\}\ \textbf{skip}\ \{P\}$$

- affectation

$$\{P_E^x\}\ x := E\ \{P\}$$

où la propriété P_E^x est obtenue en remplaçant chaque occurrence de x par E dans P.

- condition

$$\frac{\{P \wedge B\}\ S_1\{Q\},\ \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}}$$

- itération

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P \wedge \neg B\}}$$

- composition séquentielle

$$\frac{\{P_1\}\ S_1\ \{P_2\},\ \{P_2\}\ S_2\ \{P_3\},\ \ldots,\ \{P_n\}\ S_n\ \{P_{n+1}\}}{\{P_1\}\ \textbf{begin}\ S_1\ ;\ S_2\ ;\ \ldots\ ;S_n\ \textbf{end}\ \{P_{n+1}\}}$$

- conséquence

$$\frac{\{P_1\}\ S\ \{Q_1\},\ P \vdash P_1,\ Q1 \vdash Q}{\{P\}\ S\ \{Q\}}$$

La notation $P \vdash Q$ signifie que Q est prouvable en prenant P comme hypothèse. Pour les programmes parallèles, on exécute des instructions S_1, S_2, ..., S_n en parallèle en utilisant les mots clés **cobegin** et **coend**. L'exécution se termine lorsque toutes les instructions S_i sont terminées. Lorsque plusieurs processus s'exécutent de manière concurrente, il est nécessaire de pouvoir retarder l'exécution de certaines instructions pour éviter des conflits tant qu'une condition n'est pas satisfaite. On utilise dans ce cas le mot clé **await**. L'instruction **await** B **then** S, permet de retarder l'exécution de l'instruction S tant que la condition B est fausse. Ensuite l'instruction S est exécutée de manière atomique. Nous présentons les axiomes définis par Owicki et Gries pour traiter ces mots clés.

- attente

$$\frac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}}$$

- section parallèle

$$\frac{\{P_1\}\ S_1\ \{Q_1\},\ \ldots,\ \{P_n\}\ S_n\ \{Q_n\}\ \textit{sont sans interférence}}{\{P_1 \wedge \ldots \wedge P_n\}\ \textbf{cobegin}\ S_1 // \ldots // S_n\ \textbf{coend}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

La notion de non interférence est définie de la manière suivante. Soit le triplet $\{P\}\ S\ \{Q\}$ et une instruction T avec $pre(T)$ comme pré-condition, on dit que T n'interfère pas avec le triplet si les deux conditions suivantes sont vérifiées :

1. $\{Q \wedge pre(T)\}\ T\ \{Q\}$
2. Soit S' une instruction de S qui n'est pas dans un **await**, alors $\{pre(S') \wedge pre(T)\}\ T\ \{pre(S')\}$

La notion de variables auxiliaires est nécessaire afin de prouver la correction de programmes parallèles. Dans [Owicki et Gries, 1976], des exemples montrent comment utiliser ces variables. Elles sont utilisées uniquement pour la preuve de propriétés et non par le programme lui-même. Elles permettent d'enregistrer l'historique d'exécutions et d'indiquer quelle partie du programme est actuellement exécutée.
Pour vérifier qu'un programme ne sera pas bloqué, ils définissent une condition qui à partir des pré-conditions et des post-conditions des instructions permet de garantir ce résultat. L'explication est donné précisément dans [Owicki et Gries, 1976]. Ainsi, ils sont en mesure de définir une règle qui permet de prouver qu'un programme parallèle est correct. Cette règle est :

- cobegin avec terminaison

$$\frac{\begin{array}{c}\{P_1\}\ S_1\ \{Q_1\},\ \ldots,\ \{P_n\}\ S_n\ \{Q_n\}\ \textit{sans interférence} \\ \{P_1\}\ S_1\ \{Q_1\},\ \ldots,\ \{P_n\}\ S_n\ \{Q_n\}\ \textit{sans blocage}\end{array}}{\{P_1 \wedge \ldots \wedge P_n\}\ \textbf{cobegin}\ S_1 // \ldots // S_n\ \textbf{coend}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

Ainsi, Owicki et Gries ont étendu le travail de Hoare pour prouver des propriétés sur les programmes parallèles. En plus de la correction partielle, leur système permet de garantir qu'il n'y aura pas de blocage et qu'un programme terminera. Cependant, la preuve de non interférence et de non bloquage est complexe sur des algorithmes simples. Donc il paraît difficile de la mettre en œuvre sur des programmes réels.

3.2 Parallélisation de programmes en utilisant les archétypes et le raffinement

Berna Massingill [Massingill, 1998] a développé une méthodologie basée sur le raffinement qui permet de développer un programme parallèle à partir d'un programme séquentiel. Pour cela, elle utilise des outils classiques pour les programmes séquentiels (tests et *débogages*) et une technique utilisant des transformations justifiées formellement. Ses travaux poursuivent les travaux réalisés par les concepteurs d'UNITY.

3.2.1 Présentation de la méthodologie

Elle est basée sur des transformations qui sont appliquées sur une partie du code séquentiel. Ces transformations préservent la sémantique du code. Elles sont guidées par les patrons (*patterns*) fournis par la programmation parallèle avec archétypes. Elle emploie ce mot pour définir une abstraction, similaire à la conception avec patrons (*design patterns*), qui capture les similitudes d'une classe de programmes. Par exemple, le paradigme « diviser pour régner » est un archétype.

Archétypes

Les méthodes utilisant la conception par patrons commencent par identifier une classe de problèmes qui ont une structure de calculs similaire, puis créent une abstraction commune aux problèmes de cette classe. Les archétypes de programmation parallèle sont des archétypes qui s'appliquent à des programmes parallèles pour lesquels on définit une stratégie de parallélisation et une structure de communication. Ainsi, on peut disposer d'une collection d'archétypes qui sont en fait des transformations conservant la sémantique du code et que l'on peut appliquer à toutes les parties de codes qui satisfont l'archétype choisi. Ces archétypes peuvent être prouvés si on le souhaite.

Raffinement

La nouveauté de cette méthodologie réside dans les transformations que l'on applique au programme séquentiel afin d'obtenir une version parallèle de celui-ci. Idéalement, toutes les transformations devraient être établies et prouvées, mais seule la dernière utilise une technique spéciale qui a retenu l'attention de Massingill en raison de la nouveauté de cette transformation. En plus, toutes les transformations en séquentiel peuvent être testées, déboguées et validées. Le but de sa démarche est d'obtenir par raffinements successifs une version spéciale de son programme séquentiel, qu'elle appelle « version séquentielle simulant le parallèle ». Cette version spéciale simule un programme parallèle de N processus s'exécutant sur une architecture à mémoire distribuée avec envoi de messages. C'est la transformation de la « version séquentielle simulant le parallèle » à la version parallèle que Massingill a prouvée de manière rigoureuse.

« Version séquentielle simulant le parallèle »

Un programme qui satisfait cette définition présente les caractéristiques suivantes :

- Les données atomiques, c'est-à-dire qui ne contiennent plus de sous-objets, autrement dit des données scalaires[6], sont partitionnées en N groupes, un par processus simulé. Il se peut que des variables soient dupliquées, par exemple les compteurs de boucles ou les constantes.
- Les calculs sont décomposés en une séquence de blocs de calculs locaux en alternance avec des opérations d'échanges de données qui synchronisent les calculs. La décomposition doit respecter les règles suivantes :
  - Chaque bloc de calcul local est un groupe de N calculs, qui modifient uniquement les variables qui leur sont locales. Un bloc correspond à une partie du programme parallèle dans laquelle les processus s'exécutent de manière indépendante sans interaction.
  - Chaque opération d'échange de données est composée d'un ensemble d'affectations qui respectent les contraintes suivantes :
    - Si une donnée atomique est la cible d'une affectation, c'est-à-dire qu'elle est à gauche d'une affectation, elle ne doit pas être référencée dans une autre affectation.
    - Une affectation ne peut référencer (à gauche ou à droite de l'opérateur d'affectation) que des données atomiques appartenant à la même partition de calcul. Par contre, une affectation peuvent référencer des données (non atomiques) d'autres partitions.
    - Pour chaque processus simulé i, au moins une instruction d'affectation doit affecter une valeur à une des variables locales de i.

    De tels blocs correspondent à des sections du programme parallèle dans lesquelles les processus échangent des messages. Chaque instruction d'affectation peut être implantée en utilisant une communication point-à-point ou un groupe d'opérations d'envoi de messages avec le même expéditeur et le même récepteur pour plus d'efficacité.

Le programme parallèle est donc composé de N processus séquentiels déterministes. Les processus ne partagent pas de variables puisque chaque processus possède un espace d'adressage distinct. Les processus interagissent uniquement par l'intermédiaire de messages transitant sur des canaux avec le protocole un lecteur / un rédacteur. Une exécution du programme consiste à un entrelacement équitable des actions des processus. Avec cette définition, Massingill démontre ensuite que l'exécution d'un tel programme « simule » parfaitement l'exécution du programme parallèle correspondant, dans le sens où toutes les exécutions du programme parallèle donnent le même résultat que le programme séquentiel.
La figure 3.1 montre la relation existant entre la version parallèle simulée et la version parallèle réelle d'un programme.
La difficulté rencontrée avec cette méthode est donc de produire un programme séquentiel simulant un programme parallèle, car ce travail nécessite un temps important. D'autre part, cette tâche est pénible et peut introduire par mégarde des erreurs. C'est pourquoi le processus de transformations est guidé par l'utilisation des archétypes. Ceux-ci fournissent pour une classe spécifique de programmes, l'abstraction ainsi que les principes de parallélisation pour les programmes de cette classe, et encapsulent directement le code pour la parallélisation en utilisant la bonne bibliothèque. Donc, si l'archétype existe, la parallélisation du programme n'en est que plus facile.

6. Un élément scalaire d'un tableau est considéré comme une donnée atomique.

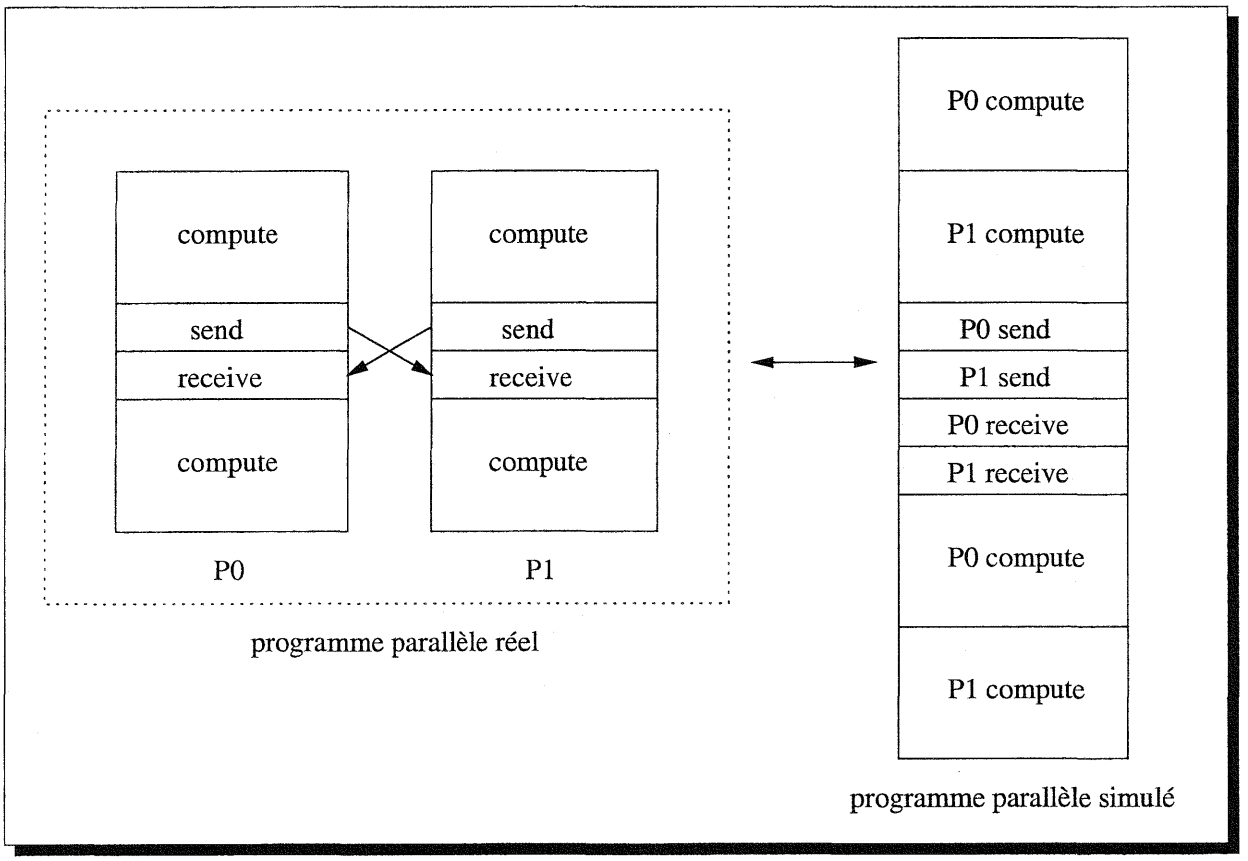


FIG. 3.1 – *Correspondance entre un programme parallèle réel et un programme parallèle simulé*

Pour exécuter la simulation du programme parallèle sur une machine séquentielle, Massingill procède de la manière suivante :

1. La simulation de l'exécution concurrente est réalisée par entrelacement d'actions des processus simulés.
2. La simulation sépare les espaces d'adressages en définissant un ensemble d'espaces d'adressages.
3. La simulation des communications est réalisée à travers des canaux représentés par des files en prenant soin de lire un canal uniquement lorsqu'il n'est pas vide.

3.3 Raffinement de programmes parallèles

Tous les travaux sur UNITY ont pour but de développer des programmes parallèles en appliquant des raffinements sur les différentes versions d'un programme. Comme nous avons déjà présenté UNITY dans le chapitre précédent (chapitre 2), nous n'allons pas en reparler dans ce chapitre. Cependant il faut garder en tête qu'UNITY a également sa place dans ce chapitre. Le choix que nous avons fait de le placer dans le chapitre précédent tient simplement au fait que nous avons voulu présenter deux modèles pour réaliser des spécifications formelles précédemment.
Le modèle que nous présentons maintenant est élaboré par Kaisa Sere. Dans sa thèse [Sere, 1990], elle explique comment on peut utiliser les systèmes d'actions puis les dériver ou raffiner pour aboutir à un programme parallèle.

3.3.1 Modéliser le parallélisme avec des systèmes d'actions

Le parallélisme est souvent formalisé à l'aide de processus où chacun correspond à un programme séquentiel. Les processus sont exécutés en parallèle et communiquent entre eux par l'intermédiaire d'envoi et de réception de messages ou à travers la mémoire partagée. Sere explique qu'un des problèmes des formalismes orientés processus réside dans le fait qu'il est difficile de raisonner sur le comportement entier du système basé sur le comportement de plusieurs processus. Il est également contraignant de vérifier formellement qu'un système dans sa globalité est correct vis-à-vis de sa spécification car les processus peuvent communiquer entre eux à tout moment. En général la vérification de tels systèmes est réalisée en deux étapes [Owicki et Gries, 1976] :

1. Il faut montrer que chaque processus isolé se comporte selon sa spécification.
2. Ensuite, il faut s'assurer que lorsque les processus sont exécutés en parallèle, il n'existe pas d'interférence entre les processus de manière incorrecte.

Son travail se différencie de cette approche en modélisant le parallélisme par des systèmes d'actions basées sur des événements. Le formalisme de systèmes d'actions est décrit dans sa version la plus récente dans [Back et Kurki-Suonio, 1988]. Il permet de décrire le comportement de programmes parallèles et distribués par l'intermédiaire d'actions que les processus du système exécutent en coopérant entre elles. Plusieurs actions peuvent être exécutées en parallèle tant qu'elles ne partagent pas de variables communes. Les actions sont atomiques, c'est-à-dire que si une action est choisie pour être exécutée, elle l'est entièrement sans aucune interférence avec les autres actions du système. L'atomicité des actions garantit que l'exécution parallèle de 2 actions a et b donne le même résultat qu'une exécution séquentielle non-déterministe $a; b$ ou $b; a$.

Présentation du formalisme

Un système d'actions parallèles $\mathcal{A}$ est un groupe d'actions de la forme :

$$\mathcal{A} = [\![\textbf{var } x; S_0; \textbf{ do } A_1 \ [] \ldots [] \ A_m \textbf{ od}]\!] : z$$

Ce groupe d'actions initialise les variables d'états y tels que $y = x \cup z$, par l'intermédiaire de l'instruction S_0. Les variables z sont des variables globales et les variables x sont locales à $\mathcal{A}$. Chaque action A_i est de la forme $g_i \rightarrow S_i$, où la garde g_i est une condition booléenne et où le corps S_i est un groupe d'instructions séquentielles. Les variables d'états référencées dans l'action A sont notées vA. On dit que deux actions A et B sont indépendantes si $vA \cap vB = \emptyset$, sinon elles se font concurrence.
Un système d'actions en séquentiel se comporte comme un bloc d'instructions gardé de Dijkstra [Dijkstra, 1976]

$$S_0; \textbf{ do } A_1 \ [] \ldots [] \ A_m \textbf{ od}$$

sur des variables d'états. L'initialisation est d'abord effectuée, après quoi la boucle **do** est exécutée, tant que des actions A_i sont activables. Une action est activable si sa garde est vraie. On dit qu'un système d'actions est terminé, si toutes les exécutions possibles des blocs d'instructions gardés ont été exécutées, c'est-à-dire que toutes les gardes sont fausses.

3.3.2 Dérivation de systèmes d'actions

La contribution de ce travail réside dans une méthodologie proposée afin de construire des systèmes d'actions qui peuvent s'exécuter sur des systèmes à base de *Transputers* (seules ces machines étaient disponibles à cette époque). La correction du système d'actions par rapport à sa

spécification est basée sur le calcul de la plus faible pré-condition de Dijkstra [Dijkstra, 1976]. Le but du raffinement est de transformer un algorithme plus ou moins séquentiel en un système d'actions qui s'exécutent de façon parallèle. Pour cela nous avons besoin de règles spéciales qui introduisent le parallélisme dans l'exécution.

Adaptation du raffinement aux systèmes d'actions

Les méthodes suivantes permettent de raffiner progressivement un programme séquentiel en vue d'obtenir un programme parallèle. Sere définit :

1. des méthodes pour transformer des blocs d'instructions en construction **do-od**. Ces règles permettent de créer des systèmes d'actions à partir de n'importe quel langage.
2. des méthodes pour changer la granularité des systèmes d'actions. En remplaçant des actions longues par plusieurs actions plus petites réalisant le même travail, la possibilité de parallélisme à l'exécution est accrue.
3. des méthodes pour fusionner des calculs séquentiels. Celles-ci permettent de tirer avantage du développement de programmes en utilisant entre autre le paradigme « diviser pour régner » et ainsi d'introduire du parallélisme en fusionnant des systèmes d'actions pour regrouper les calculs. Ainsi le calcul est composé de parties « cohérentes » de calculs intermédiaires.
4. des méthodes pour distribuer des variables. Cela permet de rendre des actions indépendantes et, par conséquent, de modifier la manière de représenter l'état du programme. Ainsi on remplace une vision centralisée du programme par une version distribuée.

Toutes ces règles énoncées informellement sont détaillées dans [Sere, 1990].

Méthodologie pour dériver un programme

Avec les méthodes précédemment décrites, il faut ensuite établir une stratégie afin d'appliquer les différentes règles dans un certain ordre tout en respectant certaines contraintes. Voici cette stratégie :

1. Il faut commencer par diviser le problème en sous-problèmes.
2. Pour chaque sous-problème, il faut définir un système d'actions qui remplisse les contraintes suivantes :
   (a) Les variables doivent être distribuées parmi les processeurs.
   (b) Il faut transformer l'algorithme en système d'actions.
   (c) Il faut adapter la granularité du système d'actions.
3. Puis il faut fusionner les systèmes d'actions obtenus par la phase 2 afin que le système d'actions résultant satisfasse le problème désiré.
4. Finalement, il peut adapter la granularité du système d'actions en fusionnant certaines actions.

Dans [Sere, 1990], trois exemples sont complétement développés. Il s'agit d'une multiplication de matrices, une élimination de Gauss et d'une ferme de processeurs. L'explication de chacun de ces exemples prend une dizaine de pages, c'est pourquoi nous ne détaillons pas d'exemple.

3.4 Méthode de Foster

La méthode de Ian Foster [Foster, 1995] est beaucoup plus pragmatique. Son livre décrit comment construire des applications parallèles. La lecture de celui-ci montre qu'il a une grande expérience

dans ce domaine. Il a acquis cette expérience en travaillant avec de nombreuses personnes sur de nombreux grands projets de calcul scientifique. L'abréviation de sa méthodologie est PCAM pour : *Partitioning, Communication, Agglomeration and Mapping*, que nous traduisons par : découpage, communication, agglomération et « distribution des tâches aux processeurs ». La figure 3.2 représente la méthodologie telle qu'il l'a décrite dans son livre.

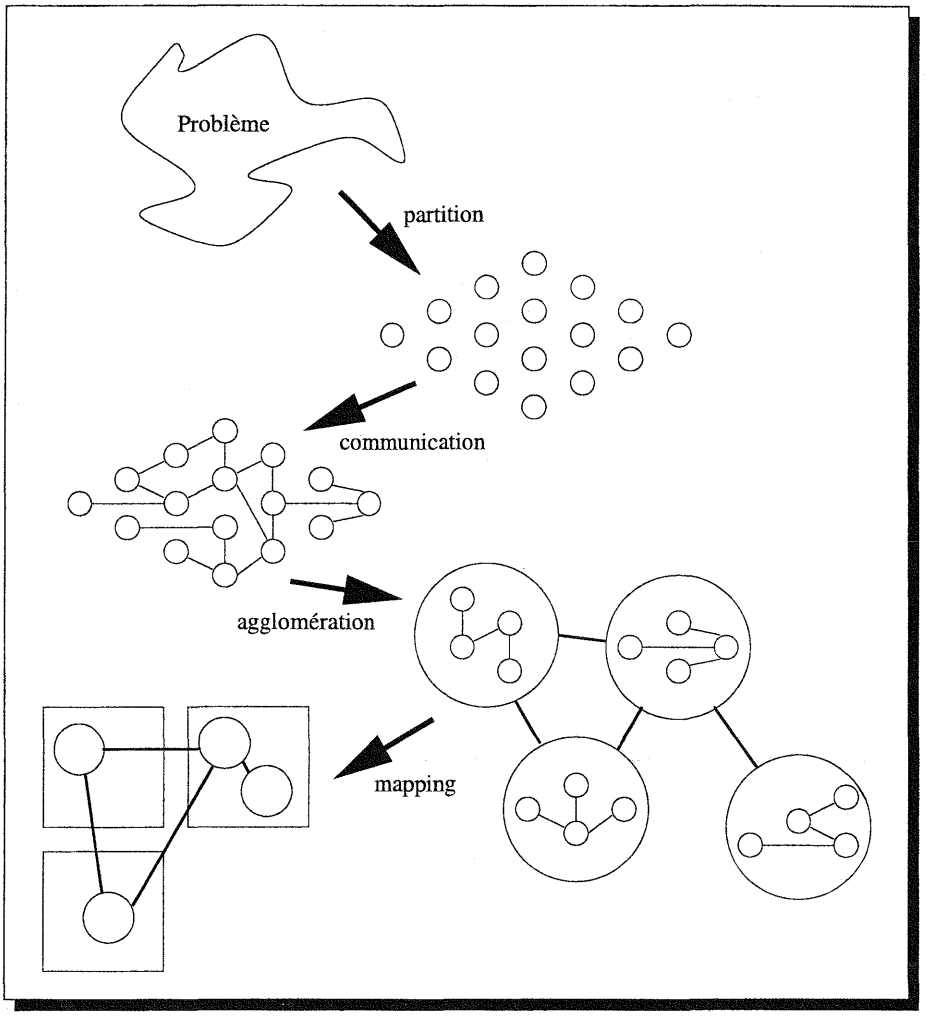


FIG. 3.2 – *La méthodologie PCAM pour le développement de programme parallèle*

Nous allons maintenant détailler les quatres étapes de cette méthodologie.

3.4.1 Création d'une partition

La première étape consiste à découper le problème initial afin d'obtenir un grand nombre de petites tâches. On peut qualifier cette étape de décomposition à grain fin. Elle a l'avantage d'offrir potentiellement le maximum de parallélisme pour le futur programme parallèle. Foster explique que, généralement, il est préférable de découper les données et ultérieurement les calculs qui sont associés à ces données. Cette technique porte le nom de décomposition de domaines. La méthode alternative qui consiste à décomposer les calculs puis les données, s'appelle la décomposition fonctionnelle. Comme elle s'intéresse à l'agencement des calculs, on constate sur certains exemples

une meilleure optimisation des calculs[7]. Ces deux techniques permettent de partitionner un problème et fournissent un moyen d'obtenir des algorithmes parallèles différents.

3.4.2 Communication

Les tâches constituées par la partition doivent pouvoir être exécutées en parallèle, mais en général elles ne peuvent pas être indépendantes. Le calcul associé à une tâche requiert souvent les données d'autres tâches. C'est pourquoi des données sont transférées pour permettre au calcul de se poursuivre. Ces informations sont élaborées dans l'étape de spécification des communications. Habituellement, on classe les communications suivant 4 axes orthogonaux : locales et globales, structuréees et non structurées, statiques et dynamiques, synchrones et asynchrones.

- Pour les communications *locales*, chaque tâche communique avec un petit nombre de processus voisins ; lors de communications *globales*, chaque processus doit communiquer avec de nombreux autres processus.
- Si les communications sont *structurées*, un processus et ses voisins forment une structure régulière telle qu'une grille ou un arbre ; les communications *non structurées* forment un réseau qui a une structure arbitraire.
- En présence de communications *statiques*, les processus échangent des données avec des processus qui sont identifiés avant l'exécution du programme ; en présence de communications *dynamiques*, les processus communiquent avec des processus qui sont fonction de l'exécution des calculs.
- Si les communications sont *synchrones*, les producteurs et les consommateurs s'exécutent de manière coordonnée ; alors que si les communications sont *asynchrones*, la consommation est différée de la production.

3.4.3 Agglomération

Les deux étapes précédentes ont constitué un algorithme qui est abstrait dans le sens où l'efficacité d'exécution n'est strictement pas prise en compte, particulièrement pour les machines parallèles. En effet, il est possible que l'algorithme soit totalement inefficace si par exemple, il crée beaucoup plus de processus que le nombre de processeurs ; surtout si la machine sur lequel il va s'exécuter n'est pas spécialisée pour traiter efficacement des tâches courtes. Cette troisième étape a donc pour but de transformer l'algorithme abstrait en un algorithme concret. Pour cela, on est amené à modifier le partitionnement et les communications proposées par les deux étapes précédentes. En général, on va essayer de diminuer le nombre de processus si celui-ci est considérablement élevé en prenant en compte les communications intensives. Cette étape modifie donc la granularité de notre algorithme.

3.4.4 *Mapping*

Cette dernière étape spécifie où chaque tâche doit s'exécuter sur la machine parallèle. Le problème d'assigner à chaque processeur les processus qu'il va exécuter n'existe pas pour les machines séquentielles et pour les machines parallèles à mémoire distribuée qui possèdent un placement automatique des tâches[8]. La difficulté de cette étape intervient lorsque l'on désire proposer un

7. Lorsque les calculs sont très complexes, la décomposition fonctionnelle permet de structurer un programme à partir de l'ensemble de ces calculs, par exemple pour simuler le climat de la terre, on peut avoir un composant pour l'atmosphère, l'océan, les glaces, les sources de CO_2, etc.

8. C'est le cas de l'Origin 2000.

mapping qui tienne compte de la « mise à l'échelle » de l'algorithme, c'est-à-dire que l'on puisse augmenter le nombre de processeurs facilement sans être obligé de modifier les trois étapes précédentes.
C'est pourquoi le but des algorithmes de *mapping* est donc normalement de minimiser les communications. Pour cela :

- On commence par placer les processus qui s'exécutent de manière indépendante sur des processeurs différents.
- Ensuite on place les tâches qui communiquent le plus fréquemment sur le même processeur afin d'accroître la localité des données.

Le problème du placement de tâches sur les processeurs est un problème *NP-complet*, cela signifie qu'il n'existe pas d'algorithme dont la complexité qui s'exprime avec une fonction polynomiale pour ce problème. C'est pourquoi on est obligé d'appliquer des heuristiques afin de donner des solutions dans un temps acceptable.

3.5 Utilisation de squelettes pour la composition

John Darlington et son équipe proposent une approche originale pour fournir une solution simple aux problèmes de programmation parallèle par composition [Darlington *et al.*, 1995]. Elle complète en quelque sorte les travaux sur PCN [Foster *et al.*, 1992]. Leur modèle a pour objectif de structurer les calculs afin de séparer les calculs parallèles des calculs séquentiels. Une telle séparation permet de se concentrer sur la coordination parallèle des composants séquentiels [Gelernter et Carriero, 1992]. L'approche de coordination pour la programmation parallèle suit une direction opposée à celle suivie par l'extension parallèle de bas niveau, non structurée, appliquée aux programmes séquentiels. En effet, elle offre un mécanisme de coordination parallèle puissant qui permet de développer des programmes parallèles avec les caractéristiques suivantes :

- ré-utilisation du code séquentiel. Les programmes parallèles peuvent être construits en composant des modules développés avec des langages conventionnels.
- ré-utilisation du code parallèle. Les programmes parallèles complexes peuvent être construits en composant des modules parallèles.
- portabilité. Les programmes parallèles peuvent être implantés sur un grand choix de machines parallèles en adaptant l'implantation des opérateurs de composition pour chaque machine.

3.5.1 SCL : Structured Composition Language

SCL signifie langage de composition structuré. Il est proposé afin de composer des procédures de haut niveau pour construire des programmes parallèles. Ce langage possède un ensemble de squelettes de haut niveau que l'on applique comme des fonctions. L'idée principale réside dans le modèle de données parallèles. Dans SCL, on réalise les calculs parallèles sur les données par l'intermédiaire d'un ensemble d'opérateurs parallèles qui agissent sur une structure de données distribués. Dans [Darlington *et al.*, 1995], les auteurs utilisent des tableaux distribuées comme structures de données parallèles. Mais ils précisent que l'on peut utiliser des données plus évoluées. Pour tenir compte de la localité, le principe consiste à spécifier pour chaque structure comment on distribue les données. Cela revient en quelque sorte à appliquer les principes de HPF [High Performance Fortran Forum, 1997].

Voici l'exemple de l'éliminination de Gauss-Seidel que nous avons repris de ce même article afin de comprendre le fonctionnement de SCL :

```
gauss A p
  = iterFor p elimPivot DA
    where
       DA = partition [column_block p] [A]
       elimPivot i x = map (UPDATE i) (applybrdcast PARTIAL-PIVOT i x)
```

L'itération principale est spécifiée par le squelette `iterFor`. `DA` est le tableau distribuée obtenu avec le squelette `partition` appliqué au tableau `A`. `PARTIAL-PIVOT` est le code séquentiel pour sélectionner le pivot. `UPDATE` est le code séquentiel qui effectue la mise à jour du pivot. La mise à jour en parallèle est spécifiée par le squelette de `map` qui est appliqué à `UPDATE` pour faire la mise à jour en parallèle. Le squelette `applybrdcast` est utilisé pour appliquer la fonction `PARTIAL-PIVOT` au $i^{\text{ème}}$ élément de `x`. Le parallélisme est donc obtenu en éliminant le pivot en parallèle dans chaque colonne du tableau distribué par l'intermédiaire du squelette `applybrdcast`.

3.6 Situation de notre travail dans ce contexte

Maintenant que nous avons présenté les outils que nous allons utiliser pour la parallélisation et les méthodes formelles et que nous avons décrit les méthodologies existantes dans ces domaines, nous pouvons situer le travail que nous avons effectué dans les deux prochaines parties. La figure 3.3 représente schématiquement ce travail.
Comme les architectures, les environnements de développement et la nature des applications sont variés, nous proposons plusieurs stratégies afin de répondre le plus précisément possible à chaque situation. Nous proposons d'utiliser un environnement de développement de preuves comme PVS pour développer la preuve d'une spécfication. Concernant la preuve de parallélisation utilisant une décomposition de domaine, nous avons construit une méthode qui fait appel à un prouveur. Et afin de générer du code depuis une spécification formelle, nous avons construit un compilateur qui prend en entrée une spécification formelle UNITY et qui produit du code fortran parallèle utilisant OpenMP.
Tout d'abord, nous avons choisi de travailler sur des applications en grandeur nature, c'est pourquoi nous avons entrepris de paralléliser trois simulations développées par des physiciens et des chimistes. Ces parallélisations ont été réalisées avec OpenMP et MPI. Lorsque nous utilisons OpenMP, notre démarche est assez différente de celle de Foster car nous effectuons de la parallélisation de boucles, même si pour la première simulation nous avons décomposé le problème. La différence est due à la mémoire partagée qui simplifie le problème des communications. Enfin le système aidé par le compilateur se charge de l'agglomération ainsi que du *mapping*. En ce qui concerne la parallélisation avec MPI, nous découpons effectivement le problème initial, puis la nature régulière du code fait que les communications ainsi que l'agglomération sont évidentes comme le *mapping* puisque nous disposons un processus par processeur. La démarche de Darlington n'est pas comparable à la nôtre puisque nous avons procédé de manière pragmatique pour ces parallélisations. De plus l'utilisation de squelette nous aurait contraint de récrire une grande partie du code, notamment pour la simulation de dynamique moléculaire. En effet, la difficulté avec un code long réside dans l'utilisation des variables globales (surtout en Fortran). Avec OpenMP, elles ne posent pas de problèmes car seules certaines boucles sont parallélisées.
Notons également que tous les travaux sur la parallélisation automatique [Collard, 1995] sont très difficilement applicables sur les simulations que nous avons parallélisées. La raison principale réside dans la longueur et la complexité de celles-ci. Pour la simulation de dynamique moléculaire,

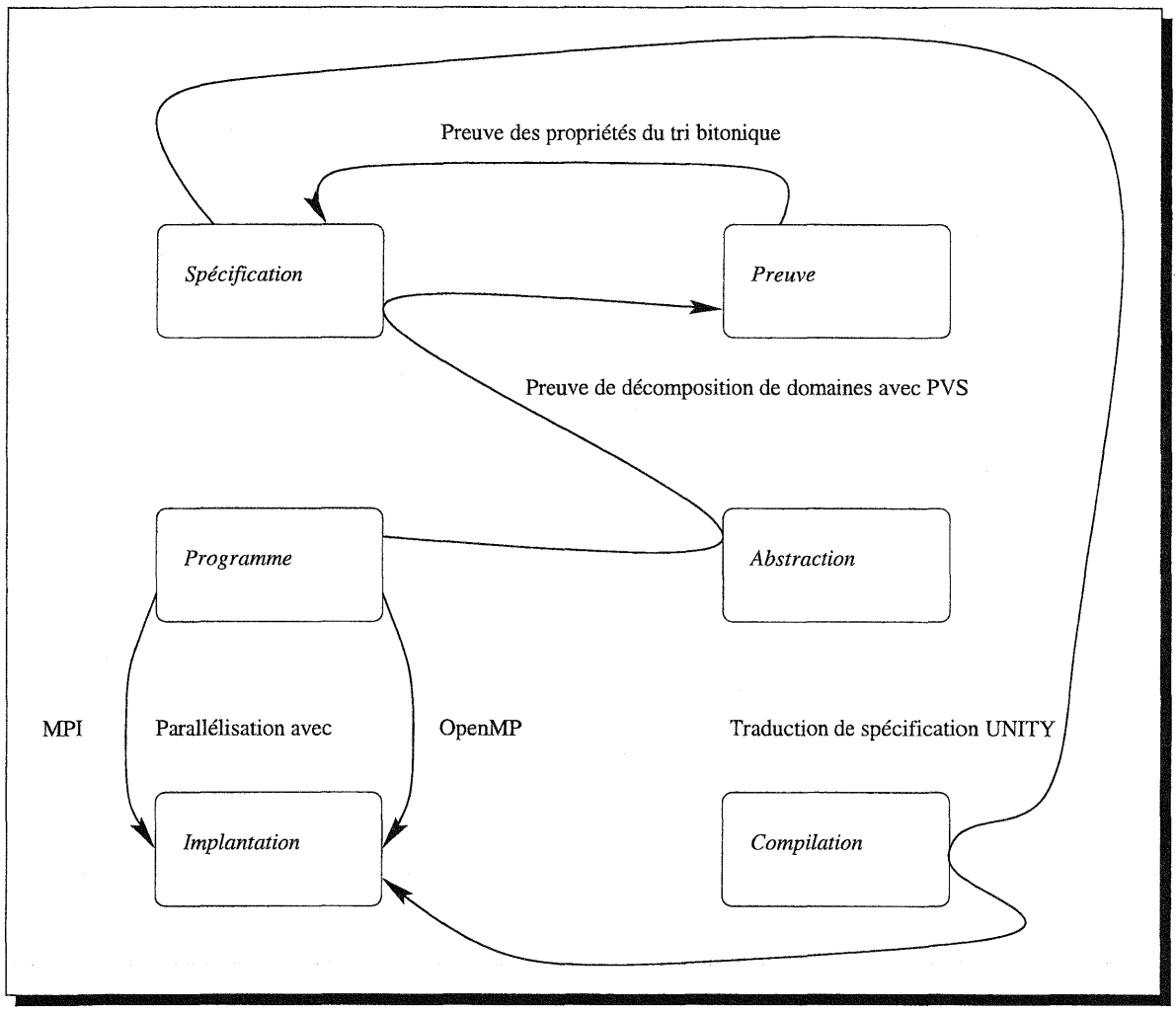


FIG. 3.3 – *Travaux entrepris*

la nature irrégulière de cette simulation est une autre raison. Néanmoins, nous avons pu utiliser la parallélisation automatique lorsque nous avons utilisé OpenMP en plus de MPI. Mais cela a été possible uniquement en raison du travail de décomposition que nous avons effectué auparavant pour utiliser MPI qui simplifie le reste de la simulation.
Ensuite, afin de montrer que les méthodes formelles se révèlent bénéfiques pour le développement d'algorithmes, nous avons montré que le tri bitonique est correct. N'ayant pas de connaissances précises sur le sujet, nous avons d'abord analysé ce tri. Puis nous avons formulé des propriétés que nous avons prouvées avec PVS. Notre choix s'est porté vers PVS parce que cet outil nous paraît adapté pour ce travail. Ce tri a déjà été prouvé par d'autres personnes mais la mécanisation de ce tri dans un environnement de preuve n'avait pas été réalisée de manière complète. Finalement, nous en proposons une version parallèle qui utilise les directives de compilation du compilateur C++.
Pour montrer qu'une parallélisation basée sur une décomposition de domaines est correcte, nous avons développé une méthode qui utilise un prouveur de théorèmes. Une telle parallélisation implique une restructuration du programme afin d'identifier des parties qui peuvent être exécutées en parallèle et afin d'élaborer un code qui assemble les résultats calculés en parallèle. Nous

commencons par établir une post-condition du programme séquentiel. Nous généralisons cette propriété afin d'établir les post-conditions des différentes parties qui s'exécutent en parallèle. Puis nous définissons la post-condition de la partie qui assemble les résultats obtenus en parallèle que nous appelons la « colle ». Pour prouver que la parallélisation est correcte, nous devons prouver que les post-conditions des parties parallèles et la post-condition de la « colle » impliquent la post-condition du programme séquentiel. Par rapport aux trois approches présentées au début de ce chapitre, nous proposons une méthode qui reprend le code séquentiel existant comme Massingill, mais ni comme Sere, ni comme Owicki-Gries. De plus, notre approche permet de prouver qu'une parallélisation est correcte par le fait que le programme parallèle va fournir le même résultat que le programme séquentiel dont il est issu, contrairement aux trois approches formelles présentées dans ce chapitre. Notre méthode permet également de découvrir la post-condition de la « colle », ce qui constitue un atout. Nous avons appliqué cette méthode aux deux simulations de transition de phase que nous avons parallélisée.
Si nous parallélisons un programme en utilisant les directives d'OpenMP et un parallélisme à grain fin, nous n'avons pas trouvé de moyen satisfaisant pour prouver qu'une parallélisation est correcte. Les modifications apportées pour paralléliser une application avec OpenMP sont toutes au niveau des itérations et la modélisation des itérations d'une boucle est très complexe surtout si nous souhaitons faire des preuves. Nous détaillons précisément pourquoi et qu'est ce qu'il faudrait faire dans le chapitre 6 à partir de la page 63.
La dernière partie de notre travail consiste à montrer qu'à partir d'une spécification formelle, nous pouvons produire un code performant qui s'exécute sur une machine parallèle. Nous avons choisi pour cela de compiler des spécifications UNITY qui diffèrent un peu de celles proposées par Chandy et Misra. La version d'UNITY que nous utilisons reprend celle proposée par Van de Velde [Van de Velde, 1994]. Notre compilateur produit du code Fortran dans lequel nous insérons automatiquement des directives de compilation OpenMP. Ainsi nous montrons que les spécifications formelles et les codes parallèles ne sont pas antinomiques. Notre but était d'essayer d'établir une démarche qui reste dans un cadre formel de la spécification à l'implémentation mais cela n'a pas été possible en raison des différences de modélisations entre les outils que nous avons utilisé. PVS est en effet très différent de la version d'UNITY que nous avons utilisé. Ainsi, nous ne pouvons pas mécaniser de preuves avec la version d'UNITY que nous avons choisi. Par contre, nous pouvons générer du code à partir de cette version, ce que nous n'avons pas réussi à faire depuis un autre langage de spécification.

Deuxième partie

Expérimentations

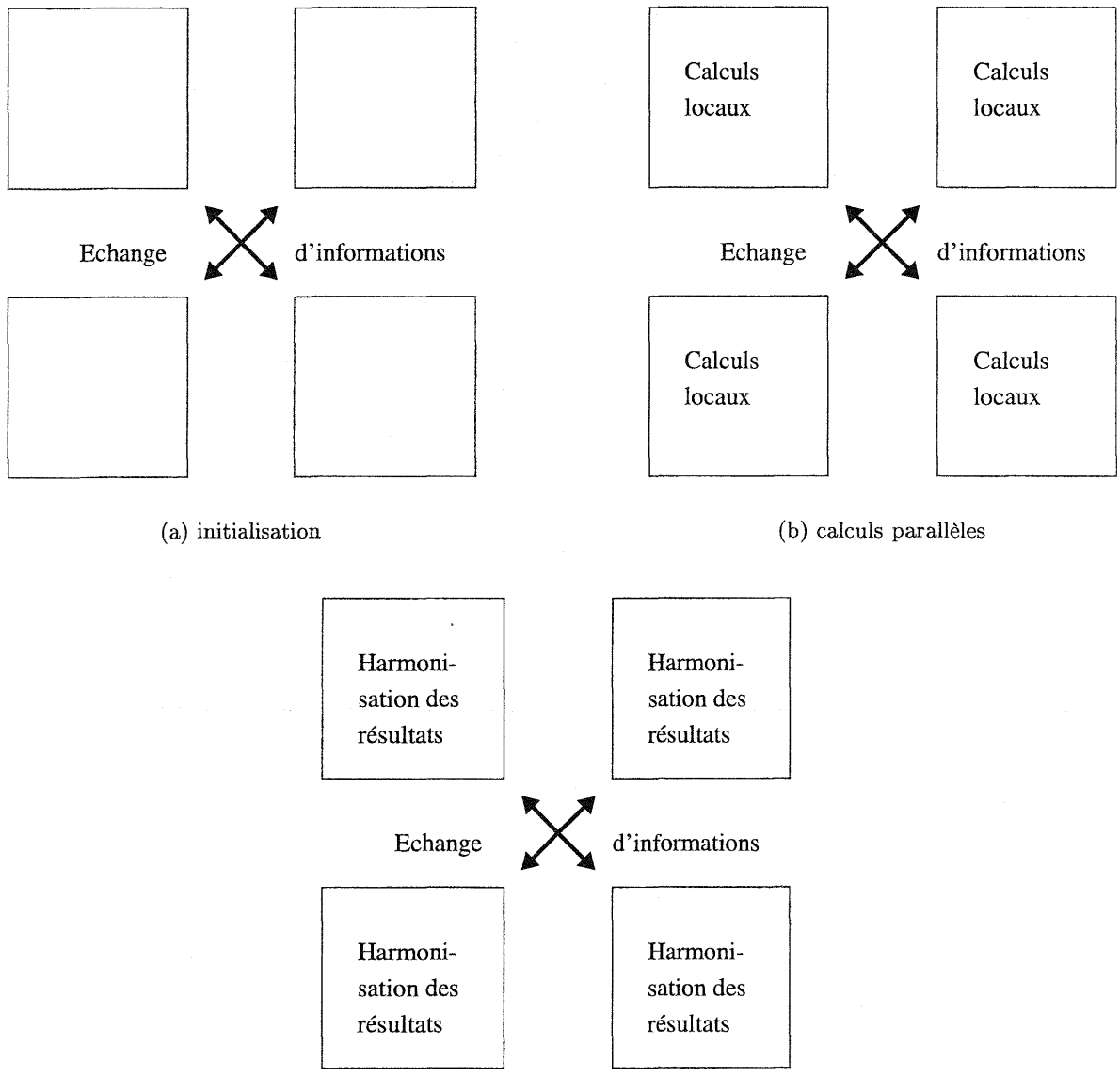
4

Archétypes utilisés pour la parallélisation

Avant de décrire les parallélisations que nous avons effectuées, nous introduisons les archétypes que nous avons utilisés. Nous pouvons distinguer deux archétypes fondamentaux qui nous ont servi lors de nos expérimentations. Le premier archétype est centré sur les données qu'il décompose. Le second archétype se base sur les traitements qu'il répartit.

4.1 Décomposition de données

La décomposition des données d'un programme consiste dans un premier temps à identifier les données que le programme utilise. Il faut distinguer les données qui sont utilisées temporairement pour parvenir à un résultat, des données qui sont essentielles pour les calculs effectués. Par exemple, les tableaux qui représentent l'état d'un système physique ou chimique sont essentiels pour une simulation. De tels tableaux, quelques soient leurs dimensions, doivent pouvoir être découpés en sous-tableaux. Un sous-tableau représente une seule partie du tableau initial dont il est issu. Ainsi les calculs des données de chaque sous-tableau peuvent être effectués sur des processeurs différents. Le domaine du tableau initial est décomposé en sous-domaines. Ensuite, suivant l'agencement des calculs, soit il faut ajouter une partie de code qui permet d'assembler les résultats s'ils ne sont que partiels, soit ils sont finaux et il ne reste plus rien à faire.
Sur la figure 4.1, nous montrons un exemple d'une décomposition qui nécessite une phase d'initialisation avant les calculs parallèles et une phase d'harmonisation des résultats après. Ces deux étapes ne sont pas tout le temps nécessaire suivant les applications. Si les données sont découpées dès le lancement du programme, certaines applications n'ont pas besoin d'une étape pour initialiser les calculs parallèles. Cette première étape d'initialisation a pour but, par exemple, que les processeurs connaissent les données, des processeurs voisins, qui sont situées sur les frontières que les processeurs ont en commun. Lorsque les calculs se déroulent, il est fréquent qu'une application requiert que certains processeurs s'échangent des informations au moins pour se synchroniser. Dès que les calculs parallèles sont achevés, certains calculs sont partiels et donc une étape d'harmonisation de ces calculs pour obtenir une solution générale est alors nécessaire. Suivant les calculs et la manière de les réaliser, il est possible qu'un processus ait un rôle de coordinateur par rapport aux autres processus. On parle alors de schéma « maître esclave ». Nous avons utilisé ce schéma pour harmoniser les résultats de notre première parallélisation.
Cet archétype de décomposition de domaines est utilisable avec plusieurs environnements de programmation parallèle. Nous l'avons utilisé avec MPI et OpenMP. Avec ces deux environne-



(a) initialisation

(b) calculs parallèles

(c) harmonisation des calculs

FIG. 4.1 – *Étapes pour la parallélisation en utilisant la décomposition de domaines*

ments, on peut noter des différences d'utilisation. Les échanges de messages sont nécessaires avec MPI. Par contre, la mémoire partagée avec les directives d'OpenMP implique que les processeurs peuvent utiliser directement les données des autres processeurs. Le travail de décomposition impose normalement qu'un processeur modifie uniquement les données qu'il possède, c'est pourquoi une décomposition de domaines est envisageable de la même manière avec OpenMP et MPI, c'est seulement l'implantation qui diffère. Mais une implantation avec OpenMP sera en général plus simple en raison de l'utilisation de la mémoire partagée qui décharge le programmeur du souci des messages. Les deux simulations de transition de phase que nous avons parallélisées utilisent cet archétype de parallélisation. La première simulation a été implantée avec OpenMP puis avec MPI. La seconde simulation a été implantée avec MPI.
Si les temps d'exécution des calculs des processus sont disproportionnés, il faut modifier la granularité des domaines, c'est-à-dire utiliser des domaines plus gros ou plus petits, ou alors il faut utiliser des domaines qui ont des tailles différentes.

4.2 Répartition des traitements

Le second archétype que nous avons utilisé consiste à répartir les itérations des boucles qui composent un programme. Pour pouvoir exécuter en parallèle plusieurs itérations d'une même boucle, il faut que l'ordre d'écriture soit conservé. Si les itérations sont indépendantes entre elles, l'ordre d'écriture est conservé. Sinon il y a deux possibilités. On peut soit utiliser des instructions atomiques qui assurent que l'exécution des instructions ne sera pas perturbée par l'exécution d'autres instructions. L'autre possibilité est d'utiliser des synchronisations ou des sections critiques afin d'assurer que seul un processus exécutera une partie du code. Mais le fait d'avoir des itérations indépendantes est généralement le meilleur moyen d'améliorer les performances. En effet, les instructions atomiques, les synchronisations et les sections critiques ont un coût d'exécution qui est important si elles sont utilisées fréquemment.

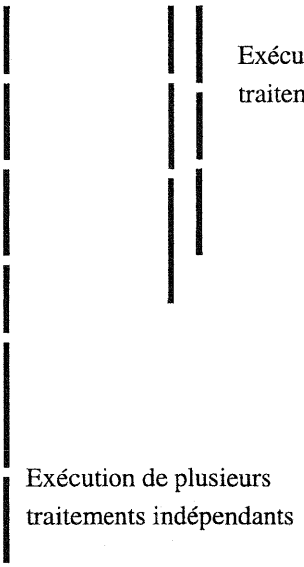


FIG. 4.2 – *Exécution de plusieurs traitement par un seul processus et par deux processus*

La figure 4.2 montre comment plusieurs traitements exécutés en séquentiel par un seul processus

peuvent être exécutés en parallèle par deux processus. Avec cette figure, on constate que le temps d'exécution n'est pas divisé par 2 avec 2 processeurs. En pratique, on rencontre souvent de telles situations car les traitements n'ont pas forcément le même temps d'exécution. Le temps d'exécution d'un traitement n'est pas toujours connu à l'avance en raison de la nature dynamique de certains traitements.
Pour remédier à ce problème, il est possible d'utiliser un ordonnancement statique lorsqu'on connaît le temps d'exécution des traitements. Sinon la solution consiste généralement à réduire la granularité des traitements en divisant chaque traitement en traitements plus petits.
Nous avons utilisé cet archétype de répartition des traitements uniquement avec OpenMP pour la simulation de dynamique moléculaire. Pour la deuxième simulation de transition de phase, nous avons utilisé l'option de parallélisation automatique du compilateur SGI qui se sert de cet archétype.

5

Parallélisation d'un système de transition de phase

Dans ce chapitre, nous décrivons la première expérimentation que nous avons menée. Il s'agit de la parallélisation d'un système de transition de phase développé par Christophe Chatelain du Laboratoire de Physique du Solide de Nancy I. Il utilise les ressources du Centre Charles Hermite pour étudier les systèmes désordonnés en physique statistique. Le but est de déterminer les propriétés critiques de systèmes modèles dans lesquels l'influence du désordre est susceptible de modifier profondément la classe d'universalité. Nous avons utilisé OpenMP et MPI pour paralléliser cette simulation en utilisant la décomposition de domaines.

5.1 Présentation du système et de sa simulation

Cette simulation est fondée sur une simulation de Monte Carlo. Mais elle diffère par rapport aux simulations de Monte Carlo « classiques » en ce sens que chaque itération nécessite le résultat de l'itération précédente, et que par conséquent on ne peut pas paralléliser cette simulation comme dans le cas d'une simulation « classique » de Monte Carlo. En fait lors d'une telle simulation, nous disposons des spins[9] sur une grille, initialement de taille 100*100 [Berche *et al.*, 1998]. Ceux-ci suivent une loi de Markov, c'est-à-dire que leur évolution d'une configuration courante à la suivante est basée sur une transition probabiliste. À chaque itération, comme le montrent les résultats de *ssrun* [Zagha *et al.*, 1996], des milliers ou des millions de nombres aléatoires sont générés suivant la taille du problème.
Nous présentons, à titre d'exemple, un extrait des statistiques fournies par cet outil *ssrun* sur une courte simulation.

```
------------------------------------------------------------------------
Line list, in descending order by function-time and then line number
------------------------------------------------------------------------
   secs      %   cum.%   samples  function (dso: file, line)
nous avons supprimé volontairement des lignes à cet endroit
 58.310   24.0    77.1      5831  SWG (test000: test000.f, 498)
 45.340   18.7    95.8      4534  SWG (test000: test000.f, 499)
  1.400    0.6    96.4       140  SWG (test000: test000.f, 503)
  1.420    0.6    97.0       142  SWG (test000: test000.f, 505)
```

9. Pour les néophytes en physique comme nous, on peut considérer qu'un spin est simplement un point.

```
0.290    0.1    97.1        29  SWG (test000: test000.f, 510)
1.820    0.7    97.8       182  SWG (test000: test000.f, 512)
2.570    1.1    98.9       257  SWG (test000: test000.f, 514)
2.100    0.9    99.8       210  SWG (test000: test000.f, 519)
```

En regardant dans le programme à quoi correspondent les lignes 498 à 519, on se rend compte qu'il s'agit de la routine qui génère les nombres aléatoires et qu'elle consomme presque 25 % du temps (de 77.1 % à 99.8 % dans la colonne 3).
Maintenant nous allons détailler les 5 étapes de chaque itération. Pour chaque étape nous donnons informellement une abstraction du programme. Notons que nous disposons d'une grille composée de spins. Chaque spin a la possibilité d'être relié à ses deux voisins horizontaux et à ses deux voisins verticaux.

1. La première étape consiste à détruire aléatoirement certains liens entre les spins de la grille. Pour cela, on parcourt l'ensemble des liens et pour chaque lien, on produit un nombre aléatoire. Suivant ce nombre, on conserve ou on détruit ce lien.
2. La seconde étape explore toute la grille et crée des graphes à chaque itération entre tous les spins connectés entre eux par un lien. L'exploration débute par un spin qui n'a pas été visité et crée à cette occasion un nouveau graphe. On attribue à ce moment une valeur aléatoire assez faible à ce graphe (dans les exemples on prend souvent une valeur comprise entre 0 et 8). Puis pour chaque voisin non visité avec lequel il existe un lien, on choisit un spin sur lequel on applique le même traitement, on lui donne le statut de « visité » et on empile les autres afin de les traiter plus tard. On répète ce processus jusqu'à ce qu'on ne puisse plus l'appliquer. Ensuite on visite tous les spins empilés en appliquant le même traitement à chaque fois. S'il ne reste plus de spin dans la pile, on choisit un autre spin non visité et on applique le même processus. Ainsi, on obtient une multitude de composantes connexes qui représentent des graphes. Chaque spin de la grille à la fin de cette étape appartient à un graphe.
3. Dans la troisième étape nous mesurons le magnétisme du système ; pour cela on parcourt la grille et pour chaque spin, on calcule la valeur du magnétisme en fonction de la valeur du graphe qui contient ce spin.
4. La quatrième étape consiste à relier par l'intermédaire de liens, deux graphes voisins et distincts, qui ont la même valeur. Puisque les valeurs des graphes sont petites et sachant qu'il y a de nombreux graphes, ce phénomène est fréquent.
5. Finalement la cinquième étape a pour but de calculer l'énergie du système, car c'est le but de la simulation. On parcourt également la grille et pour chaque spin, on établit l'énergie en fonction du nombre de liens qu'il possède.

5.2 Problèmes rencontrés pour effectuer la parallélisation de cette simulation

Avant de paralléliser cette simulation, plusieurs critères importants doivent être pris en compte. Tout d'abord, la durée d'une itération, qui est de l'ordre de la seconde[10] pour effectuer les cinq étapes citées précédemment. La génération des nombres aléatoires est un problème important. Sachant que le générateur utilisé est choisi pour ses caractéristiques, il est préférable de le conserver. En effet, les physiciens souhaitent en général avoir la plus faible corrélation possible. Ils ont

10. Pour des grilles petites, une itération est même inférieure à 100 milli-secondes.

donc choisi un générateur de type RANMAR [Vattulainen, 1994][11]. La nature de ce générateur fait qu'il est composé d'un générateur de fibonacci, qui utilise les nombres premiers 33 et 97 (pour générer un nombre n, on utilise le nombre $n-33$ et $n-97$), couplé avec une séquence aléatoire plus « classique ». Il serait souhaitable de pouvoir générer la même série de nombres aléatoires en parallèle pour chaque spin mais c'est impossible car les nombres premiers utilisés par ce générateur font qu'il n'est pas parallélisable. Il n'est pas possible de dédier un processeur à la génération de ces nombres pour deux raisons. La première est qu'il serait possible d'engendrer la même série de nombre aléatoire mais, dans ce cas, il faudrait savoir à chaque instant pour quel spin de quel processeur le nombre aléatoire est destiné et l'algorithme utilisé fait que ce n'est pas possible. La seconde raison vient du fait qu'un seul processeur n'aura pas le temps de générer assez de valeurs aléatoires si le nombre de processeurs qui consomment ces nombres est important (en séquentiel le processeur passe 25 % de son temps à générer des nombres aléatoires sur l'exemple du début de ce chapitre donc cela implique qu'au maximum on pourrait avoir 4 processeurs qui travailleraient en parallèle).
Connaissant ces contraintes, nous avons choisi dans un premier temps d'utiliser la mémoire partagée à la place de MPI [Gropp *et al.*, 1994], pour deux raisons. La première est qu'avec OpenMP, en effectuant de la parallélisation de boucles et en utilisant la décomposition de domaine, on peut travailler de manière incrémentale, c'est-à-dire que l'on peut paralléliser une partie, et une fois qu'elle est bien parallélisée, on peut se concentrer sur une autre partie du programme. La seconde raison est liée à la difficulté d'utilisation de MPI pour cette simulation. Cette difficulté vient du fait que le découpage explicite des données entrainent obligatoirement l'envoi de nombreux messages rendant la mise en œuvre de la parallélisation délicate. En effet, nous sommes obligés de modifier presque la totalité du code et il faut être très prudent afin que les messages véhiculent les bonnes données vers les bons processeurs, ce qui est source d'erreurs trop fréquentes et difficiles à détecter. Toutefois, par la suite nous avons tenté d'utiliser MPI pour paralléliser cette simulation et nous y sommes parvenu après un effort plus conséquent qu'en utilisant OpenMP.
Concernant le problème des nombres aléatoires, nous le contournons en utilisant pour chaque processeur, un générateur de nombres aléatoires RANMAR avec une graine différente. Avec le nombre d'itérations utilisées (de l'ordre du million), on constate que les résultats du programme séquentiel et du programme parallèle avec OpenMP et MPI convergent très bien.

5.3 Parallélisation de cette simulation

Le code initial est écrit en Fortran 77, c'est pourquoi nous avons conservé ce langage. La parallélisation des étapes 1 et 3 s'effectue assez simplement avec OpenMP et avec MPI, car nous pouvons décomposer la grille en sous-grilles. Les étapes 4 et 5 sont faciles à paralléliser avec OpenMP et elles demandent d'échanger les spins situés sur les frontières avec MPI.
La parallélisation de l'étape 2 requiert plus d'effort. En effet, les graphes créés à cette étape sont disposés de manière désordonnée sur la grille. Donc la décomposition de la grille en sous-grilles permet de calculer uniquement des graphes partiels. Ceux-ci ne peuvent être calculés en entier qu'avec la totalité de la grille. La solution adoptée consiste à calculer les graphes partiels, puis à coller ceux qui sont disposés sur des grilles adjacentes et qui sont reliés par un lien. On réalise cette étape de collage en attribuant à chaque graphe partiel un numéro différent et en attribuant le même numéro à deux graphes partiels reliés entre eux par un lien. Comme il est possible qu'un graphe soit disposé sur de nombreuses sous-grilles, si nous effectuons le collage en parallèle, il

11. Ce générateur a une corrélation de l'ordre de 2^{144} alors qu'un générateur classique a une corrélation de l'ordre de 2^{32}.

est possible que deux processeurs donnent deux numéros différents à un même graphe, donc nous devons soit effectuer ce travail en séquentiel, soit utiliser des sections critiques. Nous avons fait le choix de réaliser ce travail en séquentiel car les sections critiques ralentissent ce travail en parallèle. La figure 5.1 montre comment on assemble deux graphes partiels s'ils sont reliés par un lien sur la frontière les séparant. La frontière est toujours de largeur 1, elle n'est pas un recouvrement des sous-domaines. Si la frontière possède des liens, on relie les graphes qui sont à l'extrémité de chaque lien. Sur la figure nous avons représenté seulement un graphe par sous-grille, mais en réalité, il y a des graphes sur la totalité de chaque sous-grille. L'utilisation de MPI pour cette étape est plus complexe que l'utilisation d'OpenMP en raison des messages qu'il faut envoyer mais le principe est le même.

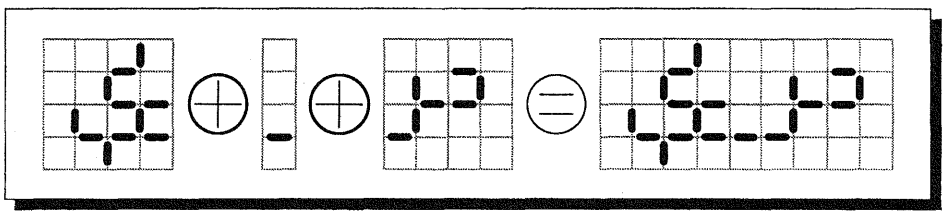


FIG. 5.1 – *Collage de deux graphes obtenus en parallèle en présence d'un lien sur la frontière*

5.4 Performances

Dans les tableaux suivants, on rassemble quelques résultats pour mesurer le gain. Le temps est exprimé en secondes. Nous avons additionné le temps de l'étape 3 et de l'étape 5 car ces deux étapes expriment le temps requis pour calculer les mesures physiques (magnétisme et énergie). Nous avons ajouté le temps nécessaire pour calculer une itération. Le temps, pour un processeur, est le temps d'exécution de l'algorithme séquentiel, utilisé initialement par les physiciens. Lorsque nous utilisons plusieurs processeurs, nous précisons le modèle de parallélisation. Tous les tests ont été réalisés avec d'autres utilisateurs sur la machine. Nous donnons les résultats avec des grilles plus grandes que 100*100 car les physiciens en ont besoin. Nous utilisons donc cette parallélisation comme un moyen de travailler sur un domaine plus grand.

Voici l'abréviation des étapes :
étape 1 : destruction des liens
étape 2 : construction des graphes
étape 3 : mise à jour des liens
étape 4 : mesures

- 10000 itérations sur une grille 200*200

nb proc	étape 1	étape 2	étape 3	étape 4	une itération	gain
1	331	282	75	42	0.073	1
4 MPI	88	112	8	19	0.023	3.2
4 OpenMP	63	91	21	16	0.019	3.8

- 10000 itérations sur une grille 600*600

nb proc	étape 1	étape 2	étape 3	étape 4	une itération	gain
1	3370	2918	732	1326	0.835	1
4 MPI	990	1169	124	240	0.252	3.2
4 OpenMP	688	733	173	215	0.181	4.61

- 5000 itérations sur une grille 800*800

nb proc	étape 1	étape 2	étape 3	étape 4	une itération	gain
1	3045	2792	718	1425	1.596	1
4 MPI	1029	1273	217	364	0.577	2.8
4 OpenMP	743	743	188	320	0.399	4
8 OpenMP	370	582	97	210	0.252	6.3
16 OpenMP	188	497	53	166	0.180	8.8

L'analyse de ces résultats révèle que l'étape où l'on a modifié le plus de code (l'étape qui concerne la création des graphes) est rapide. Il faut préciser que le seul fait de diviser la grille en sous-grilles améliore les performances en séquentiel (ceci est dû à la mémoire cache de la machine). Sur tous les tests que nous avons réalisés, l'utilisation d'OpenMP apporte de meilleurs résultats que l'utilisation de MPI. De plus, la mise en œuvre d'une parallélisation avec OpenMP est plus simple et rapide que la même mise en œuvre avec MPI. Avec OpenMP et avec 4 processeurs, le gain est proche de 4 (pour une simulation, il est même supérieur en raison de la décomposition qui favorise l'utilisation de la mémoire cache de la machine). Par contre, lorsque le nombre de processeurs croit, les performances se dégradent. La présence de deux petites parties séquentielles[12] dans l'étape 2 en sont la raison. Ces deux parties petites ne sont pas trop perceptibles avec peu de processeurs mais elles se font sentir avec 16 processeurs [Couturier et Méry, 1998b].

5.5 Conclusion de ce travail

Cette simulation de Monte Carlo a été parallélisée sans trop de difficultés en utilisant les directives de compilation d'OpenMP. Les gains obtenus sont relativement bons. Certes, ils se dégradent lorsque le nombre de processeurs croît, mais la parallélisation avec MPI n'apporte strictement rien dans ce cas. Nous avons parallélisé à titre d'essai cette simulation en MPI, mais non seulement l'effort pour le développeur se révèle plus intense, car on est obligé de découper explicitement les données, mais en plus, les résultats que nous avons obtenus sont moins intéressants. Les communications pour cette simulation doivent être effectuées très rapidement car elles sont fréquentes.
Comment assurer que cette parallélisation est correcte. La grosse difficulté dans cette parallélisation provient de la partie 2. Dans cette partie, nous calculons les graphes partiels sur chaque sous domaines. Ensuite nous collons l'ensemble des graphes obtenus. Les calculs de cette partie de la simulation et le collage ne sont pas évidents et si nous souhaitons assurer qu'ils ont été correctement parallélisés, nous devons en faire la preuve. Dans le chapitre 9, nous apportons la preuve que cette partie de la simulation est correctement parallélisée.

12. La première se trouve pour relier les graphes et la seconde pour leur assigner une valeur. En fait ces deux parties sont parallélisables mais la version séquentielle est plus rapide.

6

Parallélisation d'un système de dynamique moléculaire

Ce second travail a été mené en collaboration avec Christophe Chipot du laboratoire de Chimie Théorique de Nancy I. Il a pour but de réduire le temps de calcul de Cosmos [Owenson *et al.*, 1987], une simulation de dynamique moléculaire. Sachant que les développeurs initiaux de cette application travaillent à la NASA, l'interaction avec eux était presque inexistante et nous n'avons donc pas pu poser des questions sur le code aussi souvent que nous le souhaitions[13]. Néanmoins, nous avons pu interagir avec Christophe Chipot qui nous a sollicité pour la parallélisation. Cette fois-ci, nous avons procédé d'une manière un peu différente pour la parallélisation. Nous avons d'abord analysé le code avec *ssrun* [Zagha *et al.*, 1996] l'outil de Silicon Graphics et nous nous sommes servi des statistiques établies par le programme Cosmos. Puis nous avons parallélisé les parties les plus longues à l'exécution. Les autres sont restées intactes. Nous n'avons pas prouver que la parallélisation est correcte car nous n'avons pas trouver de méthodes et d'outils satisfaisants pour le faire. Par ailleurs les transformations effectuées sont relativement simples en raison de l'utilisation d'OpenMP. Nous expliquons quels sont les problèmes pour réaliser la preuve de parallélisations de boucles avec OpenMP. Ce chapitre montre également qu'on peut utiliser OpenMP pour paralléliser une application sans la connaître entièrement.

6.1 Présentation de la simulation de ce système

On dispose d'un système moléculaire conséquent : il est fréquent qu'il possède plus de 10000 atomes. La simulation consiste à calculer pour chaque atome l'interaction que les autres atomes exercent sur lui et réciproquement. Ces interactions nécessitent un pas d'intégration très faible (de l'ordre de la pico-seconde), afin d'intégrer les équations du mouvement de Newton en ne négligeant aucune force.
Suivant les simulations, différentes procédures interviennent afin de calculer les interactions entre les atomes d'eau, les atomes de solutés, les atomes des chaînes et les interactions entre les combinaisons possibles de ces catégories. Toutes ces procédures ont comme point commun le calcul d'une sphère de « cut-off ». Cette sphère a pour but de réduire le nombre d'interactions entre les atomes. Un atome ne peut interagir qu'avec les autres atomes qui sont dans sa sphère de « cut-off ». On considère que les autres atomes qui sont à l'extérieur de la sphère, n'ont pas d'influence sur l'atome en question en raison de la distance qui les séparent. Ainsi le nombre d'interactions se trouve réduit, mais il reste tout de même important.

13. 5 questions sur les 40000 lignes de code.

6.2 Parallélisation de la simulation

Nous n'avons pas tout parallélisé, et nous nous sommes contentés de paralléliser les procédures qui consomment le plus de temps. Ces procédures sont identifiées grâce aux outils cités précédemment, qui donnent des statistiques sur le temps d'exécution.
Le temps de calcul de chacune de ces procédures peut varier considérablement. On peut distinguer deux classes de procédures.
La première comprend celles qui sont impliquées dans le calcul des énergies et des forces. Ces procédures sont largement les plus consommatrices en temps. Leur structure possèdent des similarités dans le sens où chaque procédure parcourt les éléments du système et applique le même type de calcul pour chaque élément. Dans la version séquentielle, les interactions entre les paires d'atomes sont calculées en utilisant une liste pour stocker les atomes voisins. En se plaçant dans une perspective parallèle, des difficultés apparaissent quand un atome i interagit avec un atome j et simultanément avec un atome k. Dans ce cas, la force exercée sur i, F_i, est mise à jour à partir des contributions dues aux atomes j et k, qui ne peuvent pas s'exécuter de manière concurrente sur des *threads* différents. OpenMP possède une directive qui, en théorie, permet de règler ce problème de mise à jour concurrente. Cette directive assure que la mise à jour d'une variable est faite de manière atomique. Pour cela, il faut placer la directive `!$OMP ATOMIC` devant une instruction, par exemple `Tab(i)=Tab(i)+f`. Dans ce cas, chaque itération qui exécute cette instruction, l'exécute de manière atomique. Le problème avec cette directive est que l'instruction qui est exécutée de manière atomique doit être appelée très rarement, sinon les performances sont largement inférieures à celle du programme séquentiel, comme si on utilisait une section critique pour éviter l'écriture simultanée.
La seconde classe de procédures correspond aux opérations réalisées sur des tableaux. Comme ce type d'opérations consiste, la plupart du temps, à appliquer des transformations linéaires, la parallélisation des boucles qui régissent ces tableaux n'en est que plus aisée.
Voici un extrait du code séquentiel de la procédure qui effectue les calculs des forces entre les atomes d'eau.

```
ipoint = 0

do istk = 1,jnbstk
  npair = 0
  do jstk = 1,nbstk(istk)
    mmol = mnbstk(ipoint+jstk)
    l1 = rpw(1) - 1 + lmol
    ...
    do j = npair+1,npair+mmol
      kmol = nwlist(j,istk)
      k2 = rpw(2) - 1 + kmol
      ...
      wnrg = ...
      ...
      ra(k2,1,5) = ra(k2,1,5) + (fc1*cx1+fc4*cx4+fc5*cx5) * sofp
      ra(k2,2,5) = ra(k2,2,5) + (fc1*cy1+fc4*cy4+fc5*cy5) * sofp
      ra(k2,3,5) = ra(k2,3,5) + (fc1*cz1+fc4*cz4+fc5*cz5) * sofp
      ...
      virwwm(1) = virwwm(1) - vdx * tmpx
      virwwm(2) = virwwm(2) - vdx * tmpy
      virwwm(3) = virwwm(3) - vdx * tmpz
      ...
```

```
      ewv = ewv + wnrg
    enddo
    ...
    ra(l1,1,5) = ra(l1,1,5) + tfxo
    ra(l1,2,5) = ra(l1,2,5) + tfyo
    ra(l1,3,5) = ra(l1,3,5) + tfzo
    ...
    npair = npair + mmol
  enddo
  ipoint = ipoint + nbstk(istk)
enddo
```

Dans ce code on distingue clairement 3 boucles imbriquées. La première, qui itère sur `istk`, traite tous les *stacks* d'éléments qui constituent le système (les *stacks* centralisent plusieurs atomes, ils ont été introduits pour regrouper les calculs sur machines vectorielles). Le découpage des cellules de la simulation en *stacks* est une particularité de Cosmos afin d'éviter les vecteurs trop petits lorsqu'on exécute le code sur des machines vectorielles. La seconde boucle itère sur `jstk` et parcourt tous les atomes du *stack* `istk`. La dernière boucle qui itère sur `j`, traite les voisins des atomes `jstk`. La variable `l1` contient l'index de l'atome courant, à partir duquel l'interaction avec le voisin `k2` est calculée. `wnrg` représente l'énergie d'une interaction. Le tableau `ra` est utilisé pour stocker les caractéristiques de chaque atome: position, vitesse, accélération, etc. La troisième dimension de `ra`, lorsqu'elle prend la valeur 5, indique la nouvelle position de l'atome donné, après le pas courant de la simulation. On peut donc voir que lorsqu'on calcule la force exercée sur un atome `i`, non seulement on met à jour la position de cet atome, mais on met également à jour la position de tous ses voisins. De plus, on constate que les variables `ipoint`, `ewv` et `virwwm` sont calculées de manière itérative.
Nous présentons maintenant les modifications apportées sur le code. Comme le tableau `virwwm` a une taille limitée, on le transforme en variables (`virwwm1`, ..., `virwwm9`), ce qui rend possible une future réduction (au sens d'OpenMP). On introduit ensuite une nouvelle variable `ewv_local` afin de calculer l'énergie locale. On effectue ensuite une réduction sur cette variable. Le nombre de *threads*, `numthr`, est mis à la valeur 1. Lorsque le code est compilé en utilisant les directives de compilation, `numthr` prend dynamiquement la valeur du nombre de *threads* disponibles. On choisit ce nombre à l'aide d'une variable d'environnement utilisée par OpenMP. Comme on ne peut pas modifier de manière concurrente le même élément d'un tableau, on définit un tableau temporaire qui va contenir les valeurs modifiées localement par chaque itération. Ce tableau se nomme `tmp_ra`. On initialise ce tableau en parallèle en utilisant le nombre de *threads* disponibles afin que tous les éléments de ce tableau soient nuls. Cette initialisation est primordiale pour pouvoir effectuer une réduction sur ce tableau après l'exécution de la boucle principale.

```
virwwm1 = 0.
virwwm2 = 0.
virwwm3 = 0.
ewv_local = 0.

   numthr=1
!$ numthr = mp_numthreads()

!$OMP PARALLEL DO PRIVATE(i,j,k)
do i=1,nwat*4+nsu
  do j=1,3
```

```
    do k=1,numthr
      tmp_ra(i,j,k)=0.
    enddo
  enddo
enddo
```

Maintenant on peut paralléliser la boucle principale. La variable `ithr` est utilisée en vue de déterminer le numéro du *thread* courant qui exécute l'itération. Ce nombre est utilisé pour accéder à la valeur correcte du tableau temporaire, évitant ainsi l'accès simultané au même élément. Comme la variable `ipoint` est calculée de manière itérative, on est obligé d'ajouter quelques lignes de code afin que chaque itération ait la même valeur de `ipoint` qu'en séquentiel. Chaque accès à `ra(_,_,5)` est transformé en `tmp_ra(_,_,ithr)`. La variable `ewv` est changée en `ewv_local`, `virwwm(i)` en `virwwmi` avec `i = 1,..,9`. Les directives parallèles sont spécifiées simplement comme dans la suite de l'exemple. Le statut `SHARED` est pris par défaut par les variables pour lesquelles on n'a rien spécifié. On applique une `REDUCTION` avec l'opérateur + sur les variables `virwwm` et `ewv_local`.

```
!$OMP PARALLEL DO PRIVATE(istk,npair,ipoint,jstk,mmol,l1,kmol)
!$OMP+ PRIVATE(fc1,cx1,cy1,cz1,fc4,cx4,cy4,cz4,fc5,cx5,cy5,cz5)
!$OMP+ PRIVATE(sofp,tfxo,tfyo,tfzo,...)
!$OMP+ REDUCTION(+:virwwm1,virwwm2,virwwm3,...,ewv_local)
do istk = 1,jnbstk
  ithr=1
!$ ithr = mp_my_threadnum()+1
  npair = 0
  ipoint = 0
  if (istk.gt.1) then
    do i=1,istk-1
      ipoint = ipoint + nbstk(i)
    enddo
  endif
  do jstk = 1,nbstk(istk)
    mmol = mnbstk(ipoint+jstk)
    l1 = rpw(1) - 1 + lmol
    ...
    do j = npair+1,npair+mmol
      kmol = nwlist(j,istk)
      k2 = rpw(2) - 1 + kmol
      ...
      wnrg = ...
      ...
      tmp_ra(k2,1,ithr) = tmp_ra(k2,1,ithr) +
             (fc1*cx1+fc4*cx4+fc5*cx5) * sofp
      tmp_ra(k2,2,ithr) = tmp_ra(k2,2,ithr) +
             (fc1*cy1+fc4*cy4+fc5*cy5) * sofp
      tmp_ra(k2,3,ithr) = tmp_ra(k2,3,ithr) +
             (fc1*cz1+fc4*cz4+fc5*cz5) * sofp
      ...
      virwwm1 = virwwm1 - vdx * tmpx
      virwwm2 = virwwm2 - vdx * tmpy
      virwwm3 = virwwm3 - vdx * tmpz
```

```
      ...
      ewv_local = ewv_local + wnrg
    enddo
    ...
    tmp_ra(l1,1,ithr) = tmp_ra(l1,1,ithr) + tfxo
    tmp_ra(l1,2,ithr) = tmp_ra(l1,2,ithr) + tfyo
    tmp_ra(l1,3,ithr) = tmp_ra(l1,3,ithr) + tfzo
    ...
    npair = npair + mmol
  enddo
enddo
```

Après la boucle, il faut mettre à jour les valeurs de `ewv` et de `ra`, cette dernière étape étant réalisée en parallèle.

```
ewv = ewv + ewv_local

!$OMP PARALLEL DO PRIVATE(i,j,k)
do i=1,nwat*4+nsu
  do j=1,3
    do k=1,numthr
      ra(i,j,5)=ra(i,j,5)+tmp_ra(i,j,k)
    enddo
  enddo
enddo
```

Cet exemple de parallélisation de boucle montre bien que les modifications ne sont pas très compliquées. Cette boucle comprend tout de même 700 lignes de code, et comme toutes les autres boucles sont grossièrement de la même taille, il faut faire soigneusement attention à toutes les variables et à tous les indices de tableaux.
Pour résumer le travail à effectuer afin de paralléliser une telle boucle, il faut dupliquer les tableaux qui sont accédés simultanément en écriture, au même indice sur chaque *thread* (s'il y a trop de tableaux ou s'ils sont trop gros, ce type de parallélisation n'est pas possible). En dupliquant un tableau, il faut l'initialiser au préalable et rassembler les résultats après la boucle. Généralement ces deux parties peuvent être réalisées en parallèle.

6.3 Pourquoi nous n'avons pas effectué la preuve de cette parallélisation ?

Nous n'avons pas prouvé que cette parallélisation est correcte alors que nous proposons une méthode à cet effet dans le chapitre 9.
Afin de simplifier les explications, nous appelons version séquentielle initiale, la version du programme séquentiel que nous voulons paralléliser. Lorsque nous avons effectué des modifications sur cette version séquentielle initiale, nous obtenons une version que nous qualifions de version séquentielle en vue de la parallélisation. Les itérations des boucles de cette version du programme doivent pouvoir être exécutées de manière indépendante et lorsque toutes les itérations ont été exécutées, toutes les variables doivent avoir les mêmes valeurs que les variables après exécution de la version séquentielle initiale.
La méthode que nous proposons n'est pas adaptée à la parallélisation de boucles avec OpenMP. En effet, notre méthode demande à l'utilisateur de définir les post-conditions du programme

séquentiel et des parties parallèles. Or, en utilisant OpenMP pour paralléliser les boucles d'un programme, nous modifions légérement la version séquentielle intiale, comme nous venons de le voir dans ce chapitre, puis nous insérons les directives parallèles. Si on cherche à établir la post-condition des parties parallèles du programme, on obtient la même post-condition que pour la version séquentielle en vue de la parallélisation dans laquelle on ignore les directives de compilation d'OpenMP. Donc l'utilisation de notre méthode revient à prouver que la post-condition du programme séquentiel après modification en vue de la parallélisation implique la post-condition du programme séquentiel initial. En fait, il est équivalent de prouver que les modifications apportées pour la parallélisation sur le programme séquentiel initial fournissent un second programme séquentiel qui est un raffinement du programme initial. La preuve de raffinement de programmes séquentiels sort du cadre de notre étude et de nombreux travaux ont été effectués à ce sujet. Néanmoins, il faut distinguer les travaux qui portent sur le raffinement d'algorithmes ou de spécifications, des travaux qui portent sur le raffinement de code. La seconde catégorie est plus rare et dans le cadre de notre travail, il n'existe pas d'outil satisfaisant pour prouver que les modifications apportées à un code, dans le but de supprimer les dépendances des itérations des boucles de ce code, sont correctes.
Quelles sont les pistes possibles de recherche afin de prouver que les transformations apportées sont correctes?
Pour prouver que la version séquentielle initiale du programme est modifiée correctement, c'est-à-dire en assurant toutes les exécutions de la version séquentielle en vue de la parallélisation apportent les mêmes résultats que la version séquentielle initiale, il faut modéliser très finement les itérations des boucles. Cette modélisation doit capturer les calculs de toutes les boucles, en assurant exactement le calcul de certaines variables pour lesquelles, nous allons appliquer des transformations. Par exemple, dans l'exemple détaillé dans ce chapitre, nous avons déplacé le calcul de la variable `ipoint`. Cette modification est effectuée pour supprimer le calcul itératif de cette variable et il implique de calculer la valeur de cette variable pour chaque itération. Avec PVS nous avons tenté de modéliser cette transformation sur la boucle du programme concernée. Cela c'est révéler être un échec car la preuve nous a paru beaucoup trop compliquée par rapport à l'évidence de la transformation et donc nous avons abandonné.
Une autre piste consiste à vérifier par l'utilisation d'un *model checker* que l'exécution de la version séquentielle en vue de la parallélisation apporte le même résultat que la version séquentielle initiale. L'intérêt du *model checker* réside dans l'automatisation qu'il procure. Le point noir se trouve sûrement dans l'explosion combinatoire de tous les cas qu'il doit vérifier.
Dans tous les cas, le problème est d'assurer que toutes les boucles de la version séquentielle en vue de la parallélisation sont indépendantes. Pour cela, on peut utiliser un analyseur de dépendance. Le placement des directives de compilation d'OpenMP est simple lorsqu'il n'y a plus de dépendances entre les itérations d'une boucle. Il peut être réalisé automatiquement en analysant la portée des variables par une analyse statique. Le compilateur SGI est capable de la faire tout seul, si le travail de suppression des dépendances a été entrepris préalablement. Mais c'est ce travail qui n'est pas simple à prouver.

6.4 Résultats et conclusion

Suivant les expériences que nous avons faites et suivant les simulations, nous obtenons des résultats différents. Vu la longueur du code, nous n'avons pas tout parallélisé, et nous nous sommes contentés de paralléliser les procédures les plus longues en temps d'exécution. Dans les tableaux suivants, les temps sont exprimés en secondes. Ces tests ont été réalisés alors que d'autres calculs

tournaient sur la machine.

- 1000 itérations d'une simulation de l'inhibiteur de trypsine pancréatique des bovins dans une solution aqueuse

nombre de processeurs	temps	temps par itération	accélération
1	8503	8.5	1
4	2029	2.03	4.19
8	1205	1.21	7.05
12	987	0.99	8.61
16	832	0.83	10.21

- 1000 itérations d'une simulation de molécules de trichlorophénol en contact avec de l'eau

nombre de processeurs	temps	temps par itération	accélération
1	1850	1.85	1
4	427	0.43	4.33
8	287	0.29	6.44

- 100 itérations d'une simulation d'une bicouche de phospholipide en contact avec deux lamelles d'eau

nombre de processeurs	temps	temps par itération	accélération
1	1243	12.43	1
4	278	2.78	4.47
8	198	1.98	6.27

En analysant les résultats obtenus, on constate que les gains sont intéressants avec peu de processeurs. Pour les trois simulations, les gains sont supérieurs à 4 avec 4 processeurs. Par contre, on constate vite une dégradation lorsque le nombre de processeurs passe à 8, sauf pour la première simulation. Le gain super-linéaire est obtenu en raison des calculs qui sont localisés dans la mémoire cache de la machine, pour certaines procédures, alors qu'en séquentiel, ils ne s'y trouvaient pas.
Suivant les boucles, lorsqu'il y a peu d'itérations, il est fréquent que ce nombre ne soit pas divisible par le nombre de processeurs, par conséquent, comme le travail entre chaque itération peut varier, tous les processeurs ne terminent pas forcément leur calcul en même temps. Ce genre de situation n'est pas forcément grave, mais il ne faut pas qu'il soit trop fréquent. Une solution pour remédier à ce problème consiste à diviser le travail de chaque itération afin que le nombre d'itérations soit beaucoup plus grand que le nombre d'itérations. Une autre solution est d'appliquer une politique pour la distribution des itérations qui tienne compte de la dynamique du système. OpenMP permet de faire cela avec la directive `SCHEDULE(DYNAMIC)`.
Si on compare ces résultats à ceux obtenus avec des programmes utilisant la décomposition de domaines, ils apparaissent décevants. Mais si on compare la mise en œuvre dans les deux cas, toute considération doit être revue. Il faut quelques semaines pour paralléliser un code avec OpenMP sans forcément avoir une bonne connaissance du programme et il faut plusieurs mois pour obtenir de bons résultats en faisant de la décomposition de domaines avec MPI sur plusieurs dizaines de processeurs et dans ce cas il faut réécrire une très grosse partie du code, ce qui demande une connaissance approfondie du programme.

7

Parallélisation d'un autre système de transition de phase

De nouveau, dans cette expérimentation, il s'agit de simuler un système de transition de phase, en collaboration avec Enrico Carlon du laboratoire de Physique du Solide de Nancy I. Il travaille dans la même équipe que Christophe Chatelain. Le but de cette simulation est également de déterminer les propriétés critiques de systèmes modèles dans lequels l'influence du désordre intervient profondément. Mais cette simulation est très différente, au niveau du code et de la manière de calculer de celle présentée dans le chapitre 5 de cette même partie. Pour cette parallélisation, nous utilisons MPI. Nous pouvons également utiliser OpenMP en plus de MPI.

7.1 Présentation de la simulation de ce système

En plus des techniques de Monte Carlo qui sont généralement utilisées pour étudier les transitions de phase, une autre méthode, appelée *groupe de renormalisation de matrice dense*, a été inventée [White, 1992]. Elle consiste à engendrer un grand treillis de manière itérative, à partir d'un treillis de taille inférieure. Chaque point du treillis est une particule. Quand on calcule suffisamment d'itérations, on obtient un système très grand qui permet d'étudier les transitions de phase, phénomènes coopératifs. Chaque étape de l'algorithme consiste à déterminer la plus grande valeur propre et le plus grand vecteur propre correspondant d'une grande matrice ; les éléments du vecteur propre sont les probabilités de trouver le système dans certains états. Dans la procédure, on garde seulement les états avec la plus grande probabilité et tous les autres états sont supprimés. Par conséquent, on peut décrire un grand système avec peu d'états. Cette technique marche très bien avec des dimensions spatiales petites, mais elle est encore plus puissante que la méthode de Monte Carlo pour certaines classes de systèmes.
En pratique la matrice à diagonaliser est partagée en deux matrices, une pour la partie gauche et une pour la partie droite du système, afin de simplifier les calculs. Ces deux matrices sont appelées matrices de transfert.

7.2 Outils utilisés pour la parallélisation

Dans cette simulation, écrite en Fortran, un élément essentiel est l'utilisation de la bibliothèque ARPACK afin de calculer la plus grande valeur propre du système. Avant d'essayer de paralléliser quoi que ce soit, nous avons tout d'abord cherché l'existence d'une version parallèle de cette bibliothèque. Par expérience, si une telle bibliothèque existe, il y a de fortes chances pour qu'elle

utilise l'échange de messages pour communiquer afin qu'elle soit portable. Il n'a pas été difficile de trouver une version parallèle de cette bibliothèque basée sur MPI. Ensuite, il apparaît normal de continuer la parallélisation en utilisant MPI car toutes les données seront découpées (les tableaux en sous-tableaux et les matrices en sous-matrices). Il faut noter qu'on peut toutefois paralléliser chaque processus MPI, en utilisant des *threads* OpenMP comme nous allons le voir à la fin de ce chapitre.

7.3 Parallélisation

La première étape, pour paralléliser la simulation, a consisté à travailler avec uniquement 2 processus et la version parallèle de la bibliothèque ARPACK [Lehoucq *et al.*, 1996] (en fait la version parallèle s'appelle PARPACK). Une fois cette tâche réalisée, on obtient un programme qui s'exécute sur 2 processeurs, mais qui effectue 2 fois le même calcul, sauf pour la partie qui concerne la diagonalisation, puisque la bibliothèque s'en charge en parallèle.
Ensuite il faut paralléliser les deux routines qui ont pour charge d'effectuer la multiplication du vecteur par la matrice gauche et droite. Dans l'exemple suivant, nous avons une partie de la routine qui effectue la multiplication du vecteur par la matrice gauche. `v1` est le tableau qui contient le vecteur initial, ce tableau étant modifié par la première boucle. Dans la seconde boucle, on calcule pour chaque `iC` la valeur `prod` en utilisant `v1` et `TL` qui représentent la matrice gauche. Ensuite on affecte à `v2`, le vecteur résultat de la multiplication.

```
do ict=0,Nq**2-1,Nq+1
   do iA=1,kml
      do iB=1,kmr
         itot=iA+(ict+(iB-1)*Nq**2)*kml
         v1(itot)=v1(itot)*sqrt(zt)
      enddo
   enddo
enddo

do iB=1,kmr*Nq
   do iC=1,kml*Nq
      prod=0.d0
      iv2=iC+(iB-1)*kml*Nq
      do iA=1,kml*Nq
         iv1=iA+Nq*kml*(iB-1)
         prod=prod+TL(iA,iC)*v1(iv1)
      enddo
      v2(iv2)=prod
   enddo
enddo
```

Pour paralléliser une telle routine, nous avons d'abord analysé ces boucles. L'utilisation d'outils pour identifier les dépendances peut s'avérer intéressante. Comme il est possible de modifier l'ordre des boucles, nous l'avons fait afin de pouvoir remplacer le calcul d'`itot`, `iv1` et `iv2` par des variables intermédiaires. Ensuite, nous effectuons un appel préalable à MPI afin que chaque processeur connaisse le numéro du processeur sur lequel il s'exécute (grâce à `myid`). Pour la première boucle, nous partageons les itérations de `iB` entre les processeurs, ainsi chaque processeur a une partie des itérations à effectuer. Le calcul de `itot` est remplacé par le calcul de la

variable t. Pour la seconde boucle, les variables t et tt sont utilisées pour calculer respectivement les indices des tableaux v2 et v1, respectivement iv2 et iv1.

```
call MPI_COMM_RANK( comm, myid, ierr )

do iB=myid*kmr/nprocs+1,(myid+1)*kmr/nprocs
  do ict=0,Nq**2-1,Nq+1
    tt=Mod((ict+(iB-1)*Nq**2)*kml,kmax)+1
    do iA=1,kml
      v1(tt)=v1(tt)*sqrt(zt)
      tt=tt+1
    enddo
  enddo
enddo

t=myid*kmax+1
do iB=myid*kmr*Nq/nprocs+1,(myid+1)*kmr*Nq/nprocs
  do iC=1,kml*Nq
    tt=Mod(Nq*kml*(iB-1),kmax)+1
    prod=0.d0
    do iA=1,kml*Nq
      prod=prod+TL(iA,iC)*v1(tt)
      tt=tt+1
    enddo
    v2(t)=prod
    t=t+1
  enddo
enddo

  call mpi_allgather(v2(myid*kmax+1),kmax,MPI_REAL8,v2(1),
*      kmax,MPI_REAL8,comm,ierr)
```


Après avoir parallélisé les deux routines qui représentent le cœur de la simulation, nous avons généralisé le programme afin qu'on puisse le recompiler facilement avec un nombre de processeurs différents pour pouvoir faire des tests nombreux.
Les performances obtenues en utilisant MPI sont bonnes, mais nous avons réussi à les améliorer en utilisant OpenMP en plus de MPI. Ainsi nous pourrons comparer les deux approches.

7.4 Utilisation d'OpenMP en plus de MPI

Nous avons détaillé les avantages et les inconvénients de ces deux paradigmes en les présentant. Il est clair qu'ils n'ont pas les mêmes objectifs ni les mêmes contraintes. Néanmoins, nous allons voir qu'un programme utilisant à la fois MPI et OpenMP peut bénéficier des avantages des deux concepts. Avant de paralléliser une application avec MPI, il faut savoir comment on va découper les calculs entre les différents processeurs, et on est souvent confronté à des problèmes : ces calculs peuvent uniquement se décomposer de manière géométrique en des nombres bien particuliers. Par exemple, si on traite un cube, on le découpe souvent en n^3 sous-cubes. L'effort demandé, pour découper le programme avec MPI, est toujours requis. Ainsi, les données sont proprement découpées et les processeurs travaillent uniquement avec des données locales. Lorsqu'un processus MPI se trouve dans une phase de calcul, il est possible utiliser OpenMP pour utiliser davantage de processeurs. Chaque processus MPI s'exécute plus rapidement en raison des threads OpenMP

qui exécute le travail de ce processus MPI plus rapidement puisqu'il y a plusieurs *threads* s'exécutant sur des processeurs différents. Donc le travail d'un processus MPI au lieu d'être exécuté sur un processeur, s'exécute sur plusieurs processeurs par l'intermédaire de plusieurs *threads* OpenMP. En utilisant cette technique, la décomposition des données par MPI rend le travail de chaque processus plus simple à décomposer pour OpenMP. Cela permet éventuellement, suivant la complexité du problème, l'utilisation de la parallélisation automatique, alors qu'elle n'aurait pas forcément été envisageable, sans utiliser MPI (ou bien les résultats auraient été décevants). Donc, ainsi, un programme qui s'exécute avec m processus MPI et n *threads* OpenMP va en fait utiliser $m*n$ processus qui s'exécutent sur $m*n$ processeurs. Dans la pratique, il est préférable que m soit plus grand que n. En effet, si n est petit, le *speed-up* apporté par OpenMP est bon, et globalement on conserve un bon *speed-up* avec l'ensemble des processus.

Sur la figure 7.1, on schématise un programme parallèle qui utilise 4 processus MPI disposés sur les processeurs 2, 5, 8 et 11. Sur cette figure, nous symbolisons les communications MPI par les deux flèches horizontales. Après la première phase de communications, les calculs de chaque processus MPI sont effectués par 3 *threads* OpenMP qui s'exécutent sur les processeurs 1 à 12. En fait, pour chaque processus MPI, deux *threads* OpenMP interviennent. Comme nous considérons que le processus MPI exécutant du code parallélisé avec OpenMP devient un *thread* OpenMP, il y a bien 3 *threads* qui exécutent le travail. Dès que ces calculs sont terminés, les *threads* OpenMP se mettent en veillent et les processus MPI peuvent communiquer à nouveau.

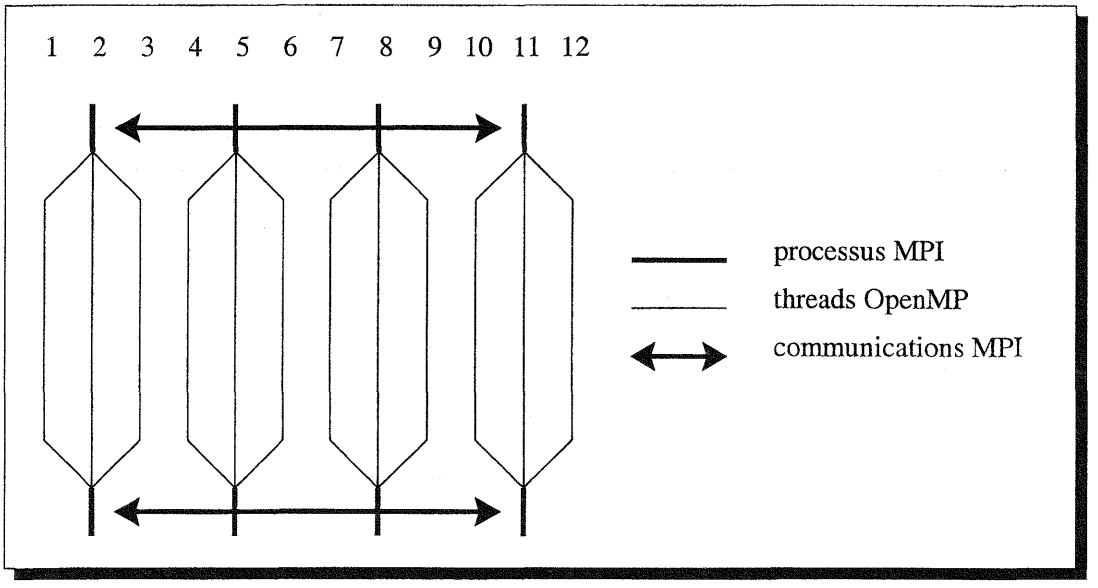


FIG. 7.1 – *Utilisation d'OpenMP en plus de MPI*

L'utilisation de MPI en plus d'OpenMP n'est pas possible et n'a pas d'intérêt car si OpenMP ne permet plus d'obtenir des performances intéressantes, ce n'est pas en utilisant des messages entre des threads que les choses vont s'améliorer. En plus, MPI nécessite que les processus soient créés dès le début, alors qu'au démarrage d'un programme utilisant OpenMP, il n'y a qu'un seul *threads*.

7.4.1 Application à notre simulation et résultats

La simulation ayant été parallélisée avec MPI, et compte tenu de la relative simplicité de ce code, nous avons pu utiliser la parallélisation automatique fournie par le compilateur SGI. Les résultats obtenus ont été réalisés en étant seul sur la machine. On exprime les temps en secondes. Pour le calcul du *speed-up*, on se réfère au programme séquentiel. Les deux programmes sont compilés avec l'option d'optimisation *-O2* et le programme utilisant MPI et OpenMP est compilé avec l'option *-apo*, ce qui permet au compilateur SGI d'effectuer de la parallélisation automatique. La bibliothèque PARPACK n'est pas compilée en *-apo* car nous avons constaté que le programme, dans ce cas, s'avérait plus lent sur les tests que nous avons réalisés. La raison est due au fait que la parallélisation réalisée par le compilateur n'est pas satisfaisante. Mais il faut préciser que cette bibliothèque est optimisée pour son utilisation en MPI. Pour ce programme, il est nécessaire que la taille de la matrice soit divisible par le nombre de processeurs MPI.

- matrice 1600*1600

nb proc (mpi)	temps	accélération	nb proc (mpi*open mp)	temps	accélération
1	15202	1			
2	8221	1.8			
4	3928	3.9			
20	723	21	10*2 = 20	710	21.4
			10*3 = 30	502	30.3
32	470	32.2	16*2 = 32	460	33
40	384	39.6	20*2 = 40	378	40.2
			10*4 = 40	355	42.8
50	510	29.8	10*5 = 50	321	47.4
			10*6 = 60	246	64.4

En voyant ces résultats, nous pouvons en conclure, tout d'abord, que nous obtenons des résultats satisfaisants simplement avec MPI. En effet, on conserve un *speed-up* linéraire jusqu'à 40 processeurs inclus. Par contre, en augmentant encore ce nombre, le *speed-up* diminue en raison des communications qui deviennent trop importantes en regard des calculs qui deviennent courts.
L'utilisation d'OpenMP plus MPI donne des résultats au moins sensiblement équivalents à MPI seulement. De plus, nous avons pu montrer qu'OpenMP offre la possibilité d'utiliser un nombre de processus, et donc de processeurs, différent de celui imposé par l'utilisation exclusive de MPI (1600 n'est pas divisible par 30 ou 60). Dans ce cas, on conserve un bon *speed-up* bien qu'une partie du programme (l'utilisation de la bibliothèque de calcul numérique) n'utilise pas OpenMP (puisque nous avons constaté que cela ralentissait le programme). Avec 40 processeurs dont 10 processus MPI décomposés en 4 *threads* OpenMP, on obtient un *speed-up* meilleur qu'en utilisant 40 processus MPI ou 20 processus MPI couplés avec 2 *threads* OpenMP. En fait ceci s'explique par le fait qu'il y a un nombre de processeurs qui maximise l'efficacité pour l'exécution de MPI et d'OpenMP. Comme les performances chutent pour OpenMP et MPI lorsque le nombre de processeurs augmente, il est préférable d'utiliser un bon compromis entre le nombre de processus MPI et le nombre de *threads* OpenMP. Le meilleur compromis avec ce programme est d'utiliser seulement 10 processus MPI. En poussant jusqu'à 50 processeurs, on observe que le gain chute un peu, mais reste largement correct avec OpenMP et MPI par rapport à la chute du gain pour 50 processus MPI. Avec 60 processeurs, dont 10 processus MPI décomposés en 6 *threads* OpenMP, on constate que le *speed-up* reste excellent, puisqu'il est de 64.

7.5 Conclusion

L'utilisation d'une bibliothèque parallèle pour paralléliser une application gagne beaucoup de temps et d'effort. Elle permet de se concentrer sur un travail qui est spécifique à une application. Il existe souvent des contraintes au niveau de la programmation ensuite, en raison de la bibliothèque de communication utilisée. La plupart des bibliothèques sont parallélisées avec MPI en raison de son utilisation massive, mais des bibliothèques utilisant OpenMP devraient apparaître assez rapidement. L'utilisation de la bibliothèque parallèle dans cet exemple a simplifié la parallélisation. Le reste des calculs est assez simple à paralléliser. Néanmoins, nous avons tenu à prouver que la parallélisation de ces calculs était correcte. Dans le chapitre 9, nous effectuons la preuve que cette parallélisation est correcte.
En outre, nous avons montré que l'utilisation d'OpenMP en plus d'une parallélisation effectuée en MPI s'avère très efficace. Le couplage de ces deux environnements devrait être plus fréquent à l'avenir car les réseaux de stations possédant des machines multi-processeurs à mémoire partagée semblent intéressants en terme de prix, de performance et de programmation.

Troisième partie

Méthodes proposées pour le développement d'applications parallèles

Dans les quatre précédents chapitres, nous avons expliqué comment nous pouvons paralléliser une application et nous avons détaillé les trois parallélisations que nous avons effectuées. Il est clair que la parallélisation d'une application avec OpenMP est plus rapide à mettre en œvre que la parallélisation de la même application avec MPI, à moins qu'il existe une bibliothèque parallèle pour MPI. Pour les deux parallélisations basées sur une décomposition de domaines, nous pouvons prouver que la parallélisation est correcte.

Les trois chapitres suivants montrent comment à partir des méthodes formelles nous pouvons développer des programmes parallèles.

Le premier chapitre expose la preuve complète d'un algorithme de tri, le tri bitonique. Le premier intérêt est de montrer qu'on peut réaliser des preuves complexes entièrement mécanisées.
Pour prouver que cet algorithme est correct, nous prouvons toutes les propriétés nécessaires à cette fin. Ainsi nous apportons la preuve de ce tri en séquentiel. En utilisant l'archétype de répartition de traitements, nous obtenons une version parallèle de ce tri en C++ avec les directives OpenMP. La traduction de la spécification en programme C++ est réalisée à la main. C'est pourquoi dans le dernier chapitre nous étudions la possibilité de générer du code parallèle automatiquement à partir d'une spécification. Le second intérêt de cette preuve a été d'acquérir la maîtrise de PVS.
Le second chapitre présente une méthode que nous proposons pour effectuer la preuve d'une parallélisation basée sur une décomposition de domaines. L'utilisation de cette méthode a pour effet de prouver que le découpage des données, plus la partie qui harmonise les résultats que nous appelons la « colle », apportent les mêmes résultats que le programme séquentiel. Nous sommes uniquement intéressés par le fait que la parallélisation soit correcte et donc nous supposons que le programme séquentiel est correct. En outre, nous pouvons également utiliser notre méthode pour « découvrir » la « colle » qui nous apportera une parallélisation correcte. Ainsi, nous offrons une stratégie permettant de concevoir le code qui harmonise les résultats pour paralléliser correctement une application.
Le troisième chapitre a pour but de montrer qu'il est possible de traduire automatiquement une spécification d'un programme parallèle en un code efficace, exécutable sur une machine parallèle. Il paraît intéressant de pouvoir obtenir rapidement, et sans effort, un code fonctionnant sur une machine parallèle, dès lors que nous avons prouvé qu'une spécification concernant le programme est correcte. Cela évite d'introduire de nouvelles erreurs avec une traduction à la main. Il aurait été intéressant de pouvoir intégrer la preuve de propriété de cette version d'UNITY dans un environnement de preuve, mais cela n'est pas possible en raison des différences existantes entre la version d'UNITY que nous utilisons et les environnements de preuve.

8

Preuve du tri bitonique avec PVS

8.1 Introduction

Dans cette partie, nous effectuons la preuve que le tri bitonique, tri facilement parallélisable, est correct. Notre but est de montrer qu'on peut, à partir de propriétés qu'on établit et que l'on prouve, obtenir la spécification d'un programme. Pour cela, nous essayons seulement d'identifier des propriétés concernant le pas d'induction d'une récursion naturelle d'un algorithme. Il est fréquent [14] de voir la preuve de cet algorithme de tri bitonique utilisant la propriété suivante : les sous-séquences d'une séquence bitonique sont bitoniques. Nous n'avons pas adopté cette solution. Dans notre preuve, nous avons besoin d'une propriété similaire, mais nous la prouvons avant de l'utiliser. Et de manière générale, nous prouvons tout ce dont nous avons besoin avant de l'utiliser de manière rigoureuse.
Pour obtenir une propriété, nous essayons d'abord d'en élaborer une idée informelle. Ensuite, nous transformons cette idée en une formule équivalente, si c'est possible, dans PVS. À cette étape, certaines idées sont supprimées car nous n'avons pas réussi à les exprimer. Puis, certaines expressions ne sont pas prouvables car elles nécessitent des hypothèses irréalistes : dans ce cas, nous reformulons la propriété jusqu'à ce que nous puissions la prouver. On peut reprocher à cette méthode de n'être pas suffisamment rigoureuse mais elle est naturelle pour aborder un problème nouveau pour lequel on n'a pas encore de solutions. Finalement, lorsque nous possédons toutes les propriétés nécessaires, nous devons vérifier que nous avons traité tous les cas. Les prouveurs, PVS dans notre cas, sont très utiles à cet effet, car ils ne procèdent pas comme l'humain, qui a « tendance » à factoriser certains cas et à en omettre d'autres supposés triviaux mais pas toujours facile à démontrer. L'intérêt de pouvoir mécaniser une preuve réside dans la confiance que l'outil procure et dans le travail répétitif qu'il peut réaliser. Nous pensons qu'il est indispensable de savoir faire une preuve à la main avant de la mécaniser dans un outil afin de guider l'outil et de pouvoir interpréter les résultats qu'il fournit.
Notre choix s'est porté pour PVS, car c'est un outil qui permet d'exprimer des propriétés avec un formalisme et une notation très proche de ceux de la logique classique. Dans le chapitre 2, nous avons vu que TLA et UNITY permettent de spécifier et de prouver des propriétés sur des systèmes concurrents. Pour le tri bitonique, nous avons besoin d'un environnement de preuve le plus puissant possible et la partie logique temporelle nous intéresse peu dans la mesure où les propriétés à prouver sont sur la correction de l'algorithme et non sur l'ordonnancement des tâches. On pourrait s'en préoccuper mais il est nécessaire de faire auparavant toutes les preuves de correction de l'algorithme sans distribution des données et des tâches.

14. À la fin de ce chapitre nous donnons des références sur ces travaux.

8.2 Problème

Dans cette section, nous expliquons le problème, et nous présentons quelques définitions. Le tri bitonique, un tri parallèle rapide [Batcher, 1968], utilise une liste bitonique que l'on définit de la manière suivante :

Définition 8.2.1 *Une liste bitonique est soit :*

- *une séquence croissante suivie d'une séquence décroissante, ou*
- *une séquence croissante, ou*
- *une séquence décroissante, ou*
- *une permutation circulaire gauche d'un des trois premiers cas.*

de 2^p nombres ($p \geq 0$)

Notons que le cas le plus simple est le premier cas (une telle liste est appelée « liste bitonique simple ») et que le dernier cas est le cas le général (une telle liste est appelée « liste bitonique générale »). Sur la figure 8.1, nous représentons à gauche une liste bitonique générale et à droite la même liste après permutation circulaire gauche des éléments. Le décalage de la permutation est égal au nombre d'éléments situés à gauche du trait en pointillé sur la figure de gauche.

FIG. 8.1 – *permutation circulaire gauche d'un des trois premier cas*

Exemple 8.2.2 *La liste (15,18,22,36,32,27,25,17,6,3,7,9) est une permutation circulaire gauche avec un décalage de 9 éléments de la liste (3,7,9,15,18,22,36,32,27,25,17,6).*

Définition 8.2.3 *Une concaténation bitonique consiste à concaténer deux listes bitoniques de 2^{p-1} nombres afin d'obtenir une liste bitonique de 2^p nombres ($p > 0$). La concaténation bitonique concatène les deux listes bitoniques de la manière suivante : on compare les nombres disposés à la place x ($x \in [0..2^{p-1}-1]$) des deux listes bitoniques et on dispose le minimum à la place x et le maximum à la place $x+2^{p-1}$ de la liste concaténée. Ainsi on obtiendra, après plusieurs étapes, des nombres rangés dans un ordre croissant. Pour obtenir une liste décroissante, après plusieurs étapes, on échange le minimum par le maximum.*

Exemple 8.2.4 *Une concaténation bitonique, qui permettra à terme d'obtenir des nombres rangés dans un ordre croissant, appliquée sur les listes (3,7,9,15,18,22) et (36,32,27,25,17,6) produit la liste (3,7,9,15,17,6,36,32,27,25,18,22) (attention cette liste n'est pas bitonique pour l'instant c'est seulement en appliquant récursivement l'algorithme qu'elle le deviendra !).*

Définition 8.2.5 *Un découpage bitonique a pour rôle de découper une liste bitonique de 2^p nombres en deux listes bitoniques de 2^{p-1} nombres ($p > 0$). Pour obtenir deux listes bitoniques de 2^{p-1} nombres, on compare les nombres disposés à la place x et $x+2^{p-1}$ ($x \in [0..2^{p-1}-1]$) de la liste bitonique de taille 2^p et on dispose le minimum à la place x dans la première liste bitonique*

et le maximum à la place x dans la seconde liste bitonique. Ainsi, après plusieurs étapes, on obtiendra des nombres rangés dans un ordre croissant. Pour obtenir des nombres dans un ordre décroissant, on inverse le minimum par le maximum.

Exemple 8.2.6 *Un découpage bitonique, qui permettra à terme d'obtenir des nombres rangés dans un ordre croissant, appliqué sur la liste (3,7,9,15,18,22,36,32,27,25,17,6) produit la liste (3,7,9,15,17,6) et la liste (36,32,27,25,18,22).*

Définition 8.2.7 *Un tri bitonique prend une séquence de nombres et constitue des listes bitoniques avec ces nombres. La taille des séquences bitoniques s'accroit étape après étape jusqu'à ce qu'on obtienne une séquence triée.*

Remarque 8.2.8 *Durant la preuve, nous verrons que l'algorithme de découpage bitonique et de concaténation bitonique sont équivalents et nous appellerons cet algorithme, l'algorithme minmax.*

8.2.1 Un petit exemple de tri bitonique

Avec un petit exemple, voir figure 8.2, on peut comprendre comment l'algorithme est appliqué. Une séquence de 2^p nombres est triée en p étapes, chaque étape p étant composée de p pas. Chaque flèche montre, suivant son orientation, si on désire obtenir une liste croissante ou décroissante à la prochaine étape ou au prochain pas. Considérons la flèche croissante sous les 8 premiers nombres de la ligne 4. Cette flèche indique que l'on veut trier ces 8 premiers nombres dans l'ordre croissant. Durant les prochains pas (lignes 5 et 6), ces nombres ne sont pas triés. C'est seulement à la fin de l'étape, plus précisément au début de la prochaine étape (ligne 7), que les nombres sont triés. Donc l'étape 3 a été effectuée en 3 pas. Chaque pas de l'algorithme consiste à concaténer une liste croissante suivie d'une liste décroissante, et ensuite à découper récursivement les listes obtenues. Cet algorithme est récursif. Tant que l'on a au moins 2 éléments dans une séquence de taille 2^p, chaque nombre x de la première (seconde) liste est obtenu en choisissant le minimum (maximum) entre les nombres x et $x+2^{p-1}$ de la liste bitonique ($x \in [0..2^{p-1}-1]$). Ensuite on applique récursivement le même algorithme avec les deux listes obtenues. On peut constater que chaque moitié de séquence a la possibilité d'être triée en parallèle puisque les comparaisons entre les éléments des deux listes sont indépendantes.
La première ligne représente la séquence non triée. Au début de l'algorithme, nous considérons toutes les paires de nombres comme des listes bitoniques avec une séquence croissante à un seul élément, suivie d'une séquence décroissante d'un élément. Avec chaque paire, on concatène les deux listes bitoniques (la croissante et la décroissante) afin d'obtenir alternativement une liste croissante et une liste décroissante de 2 nombres. La seconde ligne de la figure montre les séquences après la première concaténation. Avec la paire (13,3), on souhaite obtenir une liste croissante et avec la paire (95,5), une liste décroissante.
La prochaine étape (lignes 3 et 4) consiste à fabriquer des séquences bitoniques de 4 éléments, à partir de séquences croissantes, alternativement avec des séquences décroissantes de 2 éléments. On réalise cela, en fait, en deux pas. Pour chaque séquence, on applique l'algorithme appelé découpage bitonique (il existe deux versions de cet algorithme selon que l'on désire avoir des listes croissantes ou décroissantes).
Considérons, par exemple, la liste bitonique (3,13,95,5) (ces nombres sont les quatre premiers nombres de la seconde ligne). Maintenant on souhaite obtenir une liste croissante avec les mêmes éléments. Pour cela, on compare l'élément 1 et l'élément 3 ($3 = 1+2^{2-1}$) et on met respectivement le minimum et le maximum de ces nombres aux index 1 et 3. Donc 3 et 95 conservent la même place. En appliquant la même méthode aux index 2 et 4 (avec les nombres 13 et 5), il y a une

permutation, donc on obtient les séquences (3,5) et (95,13) (ligne 3). En utilisant la récursivité de l'algorithme, on obtient 2 séquences (3,5) et (13,95) (ligne 4). Donc la séquence initiale est, comme nous le voulions, triée.
À la ligne 4, on a alternativement des séquences croissantes et décroissantes de 4 nombres. En prenant les 8 premiers nombres, on applique l'algorithme de concaténation bitonique afin de créer une liste croissante de 8 nombres. La concaténation est réalisée à la ligne 5, en utilisant le même algorithme que précédemment. En appliquant les mêmes techniques, nous obtenons à la ligne 7, une liste croissante (de 8 nombres) suivie d'une liste décroissante, et finalement à la ligne 11, nous obtenons la séquence triée contenant tous les nombres.

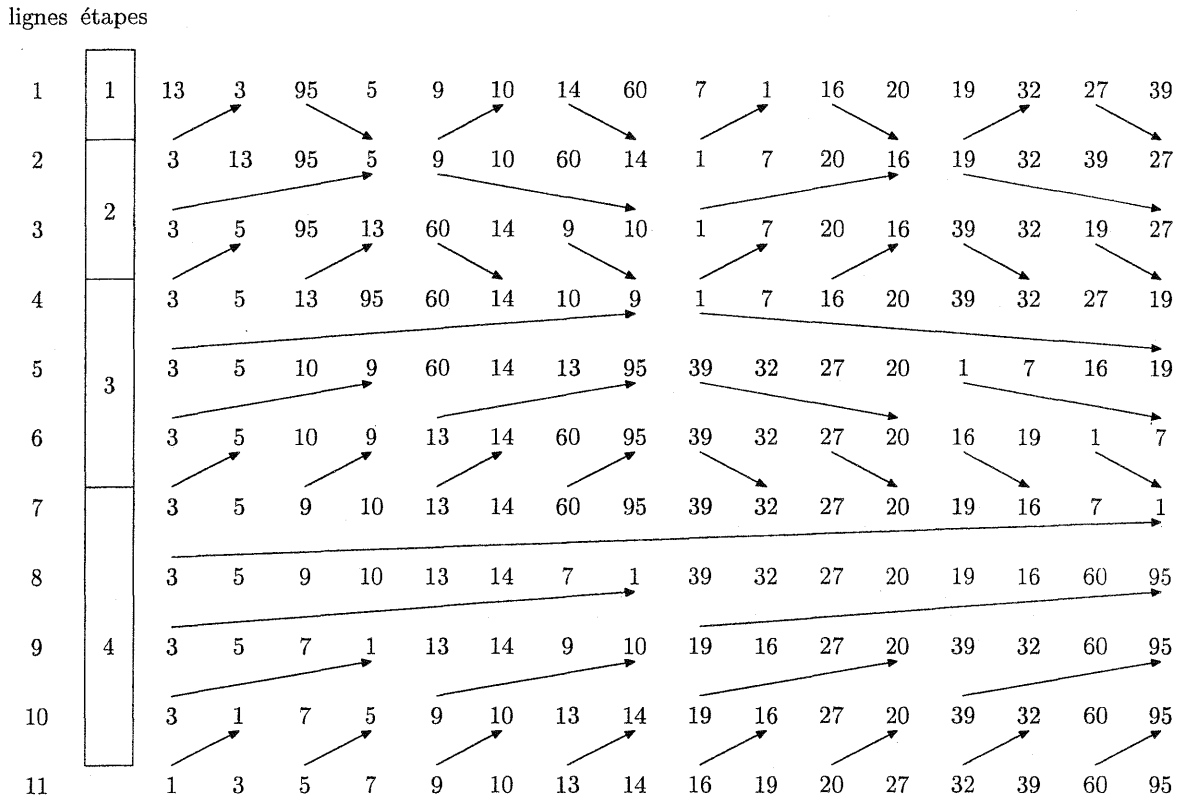


FIG. 8.2 – *Un exemple de tri bitonique*

Pour comprendre la suite, nous donnons une ébauche de l'algorithme du tri bitonique.

```
concaténation bitonique de deux listes de taille 2^(p-1)
  algorithme minmax sur la liste de taille 2^p constituée
                        des deux listes de taille 2^(p-1)
  découpage bitonique sur la premiere liste de taille 2^(p-1)
  découpage bitonique sur la seconde liste de taille 2^(p-1)
fin concaténation bitonique

découpage bitonique d'une liste de taille 2^p
  si p > 0 alors
  algorithme minmax sur la liste de taille 2^p
  découpage bitonique sur la première liste de taille 2^(p-1)
  découpage bitonique sur la seconde liste de taille 2^(p-1)
  fin si
```

`fin découpage bitonique`

Nous rappelons qu'il y a deux versions de l'algorithme *minmax* suivant que l'on désire obtenir à terme des nombres rangés par ordre croissant ou décroissant.

8.2.2 Comment prouver la correction de cet algorithme

Dans cette section, nous explicitons quelques propriétés que nous utilisons pour la preuve. Ces propriétés sont issues directement de l'exemple précédent ; mais nous allons voir qu'elles ne sont pas suffisantes pour vérifier l'algorithme.
Comme nous l'avons vu dans les **Définition 8.2.3** et **Définition 8.2.5**, il semble qu'il soit intéressant de vérifier si la concaténation bitonique de deux listes produit une liste bitonique et le découpage bitonique appliqué à une liste bitonique produit deux listes bitoniques. En fait nous allons seulement voir si l'algorithme *minmax* appliqué sur une liste bitonique produit deux listes bitoniques, puisque l'algorithme est le même pour la concaténation et le découpage bitonique. À ce stade, il faut noter, que nous ne savons pas comment prouver ces propriétés ; on peut seulement espérer qu'elles soient suffisantes afin de vérifier l'algorithme de tri bitonique.
Concernant les listes croissantes et décroissantes, nous avons pris en compte la symétrie inhérente du problème pour réduire la complexité. Il semble plus simple de prouver les propriétés requises avec des listes croissantes, et d'utiliser un algorithme pour inverser une liste croissante en une liste décroissante.
Les autres propriétés intéressantes seront introduites pendant la preuve dans la section suivante.

8.3 Preuve de propriétés intéressantes avec PVS

Dans cette section, nous expliquons le principe que nous avons utilisé afin de prouver le tri bitonique. Il faut noter que nous n'utilisons aucune propriété particulière pour la preuve de cet algorithme. La spécification qui contient toutes les propriétés que nous avons prouvées est donnée dans l'annexe C.

8.3.1 Preuve de propriétés simples

Nous présentons maintenant quelques définitions de type que nous utilisons pendant la preuve. Nous avons besoin d'un tableau :

```
Arra : TYPE = [nat->nat]
x,y,i,j,nb,nbp,low,hi:VAR nat
A,Ap: VAR Arra
```

La première ligne définit un nouveau type `Arra`, un tableau (on n'utilise pas le mot *Array*, car c'est un mot clé de PVS). La seconde et la troisième lignes définissent des variables. Donc `x`, `y`, `i` et `j` sont des variables que l'on utilise avec `FORALL` et `EXISTS` comme variables instanciées par des quantificateurs universels. `nb` est la taille du tableau courant et `nbp`[15] représente la moitié de `nb`. On considère que, après avoir découpé un tableau de taille `nb`, on obtient 2 tableaux de taille `nbp` avec `nbp=nb/2`. `hi` et `low` sont respectivement les bornes inférieures et supérieures de notre tableau. `i` est le pivot dans le tableau. On le définit de la manière suivante : avant lui, les éléments sont rangés par ordre croissant et après lui, les éléments sont rangés par ordre décroissant. `j` est un autre pivot que nous expliquerons ultérieurement. `A` et `Ap` représentent respectivement le tableau à l'état courant et à l'état suivant.

15. p est une abréviation pour prime, qui représente la même variable dans l'état suivant.

Avec ces définitions de base on définit :

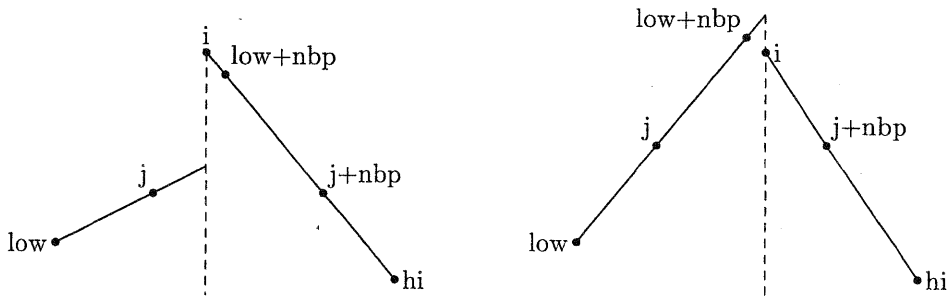
```
  IncreasingList(A,low,hi) : bool =
FORALL x,y:x>=low and y<=hi and x<=y => A(x)<=A(y)

  DecreasingList(A,low,hi) : bool =
FORALL x,y:x>=low and y<=hi and x<=y => A(x)>=A(y)
```

Ces définitions sont des formules ou des fonctions pour PVS. Elles retournent une valeur booléenne. La première, par exemple, est interprétée de la manière suivante : Pour tout `x` et `y` tels que `x` soit supérieur ou égal à `low`, que `y` soit inférieur ou égal à `hi` et que `x` soit inférieur ou égal à `y`, cela implique que la valeur de `A` à l'index `x` est inférieure ou égale à la valeur de `A` à l'index `y`.
Ensuite on définit le cas simple de la liste bitonique (les autres cas seront considérés ultérieurement) :

```
  BitonicList(A,low,hi,i) : bool =
i>low and i<=hi and IncreasingList(A,low,i-1) and DecreasingList(A,i,hi)
```

Donc nous avons une liste croissante de `low` à `i-1` et une liste décroissante de `i` à `hi`.



(a) $i \leq low+nbp$ (b) $i \geq low+nbp$

FIG. 8.3 – *Exemples de listes bitoniques simples*

Maintenant nous pouvons définir notre pivot `j` :

```
  Pivot(A,low,hi,nbp,j) : bool =
j>low and j<=low+nbp-1 and A(j-1)<=A(j-1+nbp) and A(j)>=A(j+nbp) and
(FORALL x:x>=low and x<=j-1 => A(x)<=A(x+nbp)) and
(FORALL x:x>=j and x<=low+nbp-1 => A(x)>=A(x+nbp))
```

Donc `j` est un index entre `low+1` et `low+nbp-1`, tel qu'il divise une liste bitonique simple avec les caractéristiques suivantes. Les éléments compris entre `low` et `j` sont inférieurs aux éléments compris entre `low+nbp` et `j+nbp`, et les éléments compris entre `j+nbp` et `low+nbp` sont supérieurs aux éléments compris entre `j+nbp` et `hi` (voir figure 8.3). Sans les deux dernières lignes (FORALL x ...), nous pouvons trouver des situations indésirables comme celles qui sont illustrées dans la figure 8.4. La situation de cette figure est indésirable car `j` doit être dans la partie croissante par définition et ce n'est pas le cas.

Remarque 8.3.1 *On représente une liste croissante (décroissante) par un segment ascendant (descendant), mais on ne suppose pas que les nombres sont uniformément croissants (décroissants). Ces schémas sont simplement des abstractions visuelles de séquences bitoniques : c'est*

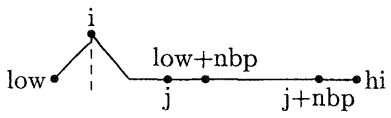


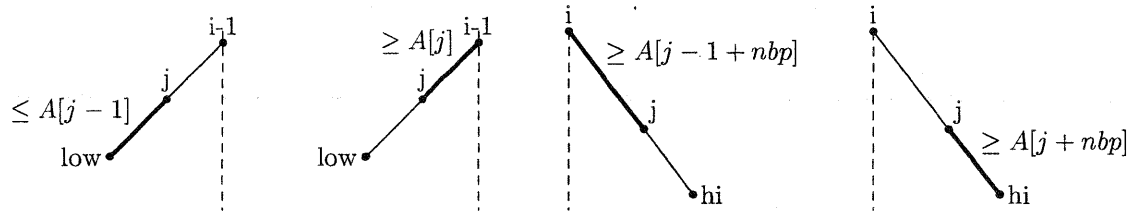
FIG. 8.4 – *Situations indésirables*

pourquoi nous ne donnons pas d'échelle. Par exemple, les séquences (4,5,12,30), (4,20,28,30) et (4,10,20,30) ont la même abstraction visuelle.

Remarque 8.3.2 *Dans une précédente version de cette spécification, nous avions oublié les deux lignes commençant par (FORALL x ...) dans la définition de* `Pivot`*. En essayant de prouver les théorèmes suivants avec PVS, nous ne pouvions pas les amener à terme car PVS ne fait pas l'hypothèse naturelle humaine que j est un nombre de la liste croissante. Donc, sans un tel prouveur de théorème, notre spécification aurait contenu une hypothèse humaine implicite.*

Maintenant que nous avons défini `i` et `j`, nous pouvons prouver les propriétés sur les places de `i` et `j`. Nous utilisons les théorèmes suivants dans la preuve :

```
    TRY_0: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>  FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)
```



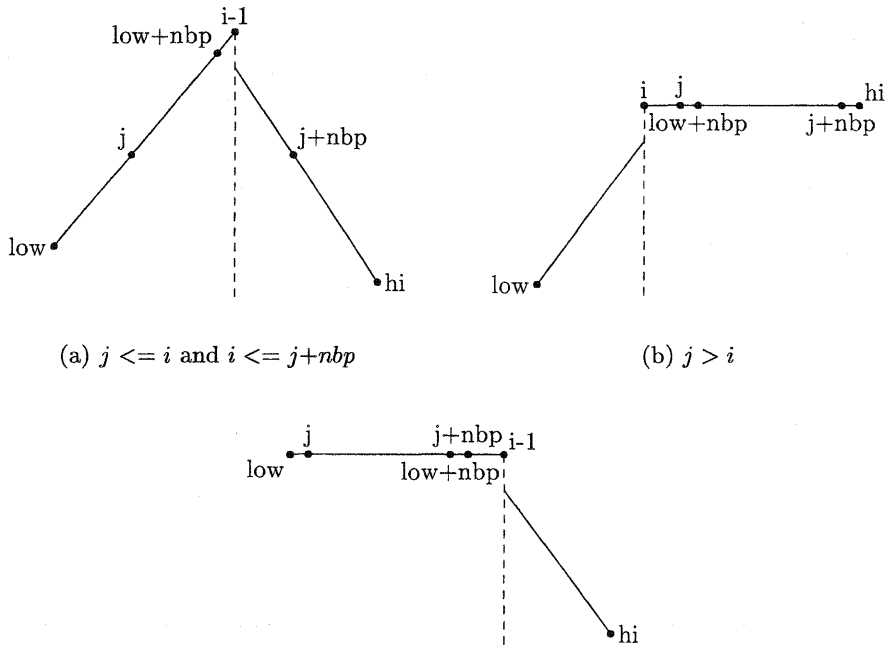
(a) `TRY_0` (b) `TRY_1` (c) `TRY_2` (d) `TRY_3`

FIG. 8.5 – *illustration des théorèmes* `TRY_0` *à* `TRY_3`

```
    TRY_5: LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
( (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
  (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) )
=>  (i>=j or ( A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i))
```

La figure 8.5 montre les propriétés des théorèmes `TRY_0`, `TRY_1`, `TRY_2` and `TRY_3`. Nous énonçons seulement le théorème `TRY_0`, car les autres sont similaires. Le trait en gras montre l'ensemble des `x` concernés, et la légende voisine indique la propriété du théorème. Par exemple, la figure 8.5(a) montre que pour tout x tel que $x \geq low$ et $x \leq j-1$, alors $A[x] \leq A[j-1]$ (c'est exactement ce que l'on a spécifié dans le théorème `TRY_0`).

Nous construisons ensuite le théorème `TRY_4` qui est la conjonction des théorèmes `TRY_0`, `TRY_1`, `TRY_2` et `TRY_3`. Nous construisons un théorème `TRY_6`, similaire au théorème `TRY_5`, dans lequel nous mettons
`i<=j+nbp or (A(i-1)=A(i-nbp-1) and A(i)<=A(i-1) and i>j+nbp`
à la place de
`i>=j or (A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i.`
Les théorèmes `TRY_5` et `TRY_6` rendent explicites les différentes places possibles de `i` et `j` (voir figure 8.6).



(a) $j <= i$ and $i <= j+nbp$

(b) $j > i$

(c) $j+nbp < i$

FIG. 8.6 – *Les différentes places de i et j*

Ensuite nous spécifions que l'algorithme *minmax*, représenté par les formules `BitonicMin` et `BitonicMax`, joue le rôle à la fois de concaténation et de découpage bitonique (dans 8.2.1 nous avons vu qu'ils étaient équivalents).

```
BitonicMin(A,Ap,low,nbp) : bool =
   FORALL x: x>=low and x<low+nbp => Ap(x) = min(A(x),A(x+nbp))

BitonicMax(A,Ap,low,nbp) : bool =
   FORALL x:  x>=low and x<low+nbp => Ap(x+nbp) = max(A(x),A(x+nbp))
```

8.3.2 Preuve que les listes obtenues sont bitoniques

La plupart des preuves suivantes sont réalisées de manière inductive sur l'algorithme récursif. Donc nous devons prouver qu'un seul pas est correct. Dans tous les cas, nous considérons que nous avons une liste de taille `nb` et l'induction nous donne deux listes de taille `nbp=nb/2`. Nous arrêtons cette induction lorsque `nbp` est égal à un.

Maintenant, nous pouvons essayer de prouver que la première liste obtenue, par l'algorithme *minmax* appliqué sur le cas simple d'une liste bitonique, est aussi une liste bitonique. En utilisant le théorème prouvé précédemment en hypothèse, nous désirons en conclure qu'il existe une nouvelle valeur j dans la liste résultante de l'algorithme *minmax* telle que j soit le pivot de la liste bitonique obtenue. Les figures 8.7 et 8.8 montrent les 2 différents cas obtenus après l'application de l'algorithme *minmax*.
Tout d'abord nous avons besoin d'une autre formule spécifiant toutes les différentes positions relatives de i et j. Cette formule est la conjonction des résultats des théorèmes TRY_5 et TRY_6 :

```
  Place_ij(A,i,j,nbp): bool =
i>0 and j>0 and
(i>=j or (A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i)) and
(i<=j+nbp or (A(i-1)=A(i-nbp-1) and A(i)<=A(i-1) and i>j+nbp ))
```

Donc le théorème suivant prouve qu'avec les hypothèses sur les places de i et j, la première liste obtenue par l'algorithme est une liste bitonique simple :

```
  BIT_0 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:
  BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
  (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
  Place_ij(A,i,j,nbp)
) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,hi,nbp)
=>
  EXISTS j:BitonicList(Ap,low,low+nbp-1,j)
```

Afin d'obtenir une bonne représentation du problème, nous découpons la séquence en 4 parties, A, B, C et D telles que A = [low, j-1], B = [j, low-1+nbp], C = [low+nbp, j-1+nbp] et D = [j+nbp, hi].

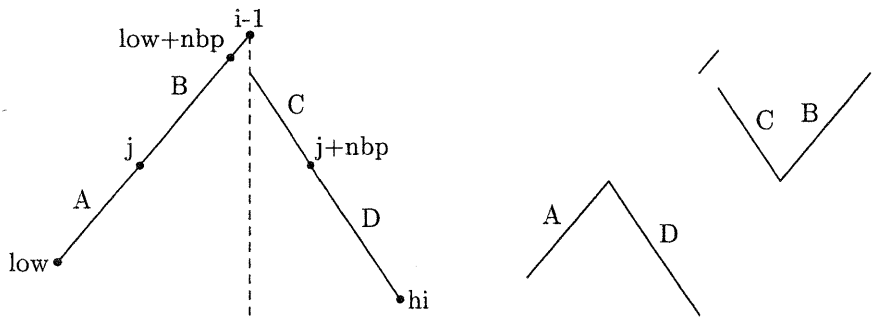


FIG. 8.7 – *Liste bitonique simple avec low+nbp <= i*

La preuve du théorème BIT_0 s'avère longue, non en raison de la difficulté, mais en raison des expansions des formules et des instanciations à résoudre qui nécessitent du temps.
Nous devons maintenant prouver que la seconde liste obtenue avec l'algorithme *minmax* est également une liste bitonique. En regardant les figures 8.7 et 8.8, les séquences formées par les ensembles C et B ne semblent pas être des séquences bitoniques simples. Dans la **Définition 8.2.1**

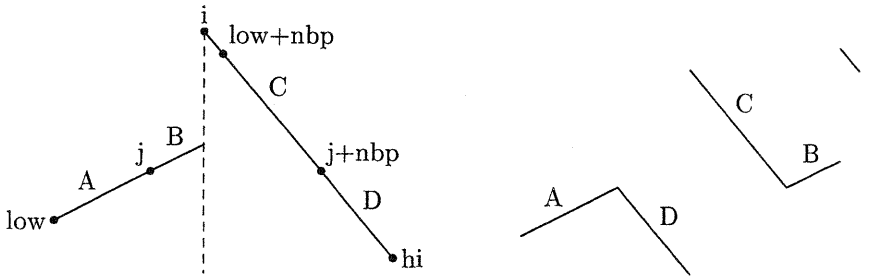


FIG. 8.8 – *Liste bitonique simple avec low+nbp >= i*

nous avons vu qu'une séquence bitonique pouvait être une permutation circulaire gauche d'une liste bitonique simple, c'est le cas général. Le théorème `BIT_1` prouve que la seconde liste obtenue après l'algorithme est une permutation circulaire d'une liste bitonique simple.

```
  BIT_1 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:
  BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
  (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
  s=j-low and
  Place_ij(A,i,j,nbp)
) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
=>
  EXISTS j:BitonicList2(Ap,low,low+nbp-1,j,nbp,s)
```

Dans ce théorème nous utilisons de nouvelles variables :

```
s,nx,ny:VAR nat
```

et des nouvelles formules :

```
  IncreasingList2(A,low,hi,nbp,s) : bool =
FORALL x,y:x>=low and y<=hi and x<=y =>
  FORALL nx,ny: nx=IF x+s>=low+nbp THEN x+s ELSE x+s+nbp ENDIF and
        ny=IF y+s>=low+nbp THEN y+s ELSE y+s+nbp ENDIF
=> A(nx)<=A(ny)

  DecreasingList2(A,low,i,hi,nbp,s) : bool =
FORALL x,y:x>=i and y<=hi and x<=y =>
  FORALL nx,ny: nx=IF x+s>=low+nbp THEN x+s ELSE x+s+nbp ENDIF and
        ny=IF y+s>=low+nbp THEN y+s ELSE y+s+nbp ENDIF
        => A(nx)>=A(ny)

  BitonicList2(A,low,hi,i,nbp,s) : bool =
s>=0 and s<nbp and (( i>low and i<=hi and
    IncreasingList2(A,low,i-1,nbp,s) and DecreasingList2(A,low,i,hi,nbp,s) )
    or DecreasingList2(A,low,low,hi,nbp,s) or IncreasingList2(A,low,hi,nbp,s) )
```

La variable `s` représente le décalage d'une permutation circulaire gauche, `nx` et `ny` représentent les nouvelles valeurs de `x` et `y`, les variables quantifiées dans le `FORALL`. Les formules `IncreasingList2` et `DecreasingList2` sont les nouvelles définitions de la liste croissante et de la liste décroissante dans lesquelles on permet des permutations circulaires gauches de leurs éléments. Si `x` plus le décalage `s` est supérieur ou égal à la moitié de la liste initiale (`low+nbp`), alors la nouvelle valeur de `x`, `nx` est égale à `x+s` sinon `nx` est égal à `x+s+nbp`. Dans les deux cas, `nx` a une valeur dans la seconde moitié de la liste, puisque seulement la seconde liste obtenue peut être une permutation circulaire gauche d'une liste bitonique simple.
Les nouvelles définitions des listes croissantes et décroissantes impliquent une nouvelle définition de la liste bitonique (`BitonicList2`). À chaque étape ou chaque pas, la valeur exacte du décalage est `s=j-low`.

Remarque 8.3.3 *Dans une précédente version de la spécification, nous avions oublié de spécifier la dernière ligne de la définition de `BitonicList2`. En essayant de prouver le théorème `BIT_1`, nous étions confrontés à un problème car nous n'avions pas considéré les cas où la seconde liste pouvait être seulement croissante ou décroissante.*

Cette preuve est de loin la plus difficile à réaliser en raison des nombreux cas différents à envisager. Voici les différents cas que nous avons dus considérer. Les positions possibles de `i` et `j` donnent trois cas (voir figure 8.6). La nouvelle définition de la liste bitonique que nous utilisons, (`BitonicList2`), capable de traiter le cas spécial de la permutation de la liste bitonique, nous donne trois cas : une liste croissante suivie d'une liste décroissante, une liste seulement croissante ou une liste uniquement décroissante. Les deux cas non classiques (seulement croissante ou décroissante) sont respectivement obtenus lorsque `i=j+nbp` et `i=j`. Ces deux cas ne sont pas vraiment différents des cas non classiques obtenus selon les positions de `i` et de `j` (lorsque: `j>i` et `j+nbp<i`) mais nous devons néanmoins les prouver. Dans le cas d'une liste bitonique simple, nous devons prouver que nous obtenons une liste croissante suivie d'une liste décroissante, donc cela nous donne deux cas de plus. Ayant une liste croissante ou une liste décroissante, il existe trois différents cas selon les valeurs de `nx` et `ny`. Ces cas sont `nx=x+s and ny=y+s`, `nx=x+s and ny=y+s+nbp`, et `nx=x+s+nbp and ny=y+s+nbp`. Au total, la combinaison de tous ces cas donne 54 combinaisons.
Nous sommes confrontés à une autre difficulté pour effectuer cette preuve. Elle est due au fait que chaque sous-but terminal (pour lequel il n'existe pas d'autres sous-buts directement prouvables) contient en moyenne entre 30 et 40 formules dont les trois quarts sont des équations ou des inéquations. Dans cette situation, PVS n'est pas capable de prouver ces sous-buts avec une commande simple. Donc nous avons été contraints d'aider PVS pour vérifier la validité de ces sous-buts en lui spécifiant quelles formules utiliser. Une preuve effectuée à la main nécessite le même effort, et en plus, on peut faire des erreurs qu'un prouveur contrôle et refuse.

8.3.3 Preuve qu'une permutation d'une liste bitonique est équivalente à une liste bitonique de l'algorithme *minmax*

Nous venons juste de voir que la seconde liste obtenue à partir de l'algorithme *minmax* est une permutation circulaire gauche d'une liste bitonique. Maintenant nous devons prouver que l'application récursive de l'algorithme sur une liste bitonique, et sur une permutation circulaire gauche d'une liste bitonique, donne le même résultat. Pour cela, nous prouvons que quel que soit le pas de récursion, si le décalage se trouve supérieur à la taille de la liste obtenue, la prochaine valeur du décalage est réduite de la taille de la liste. Cette méthode revient à dire que la nouvelle valeur du décalage est égale à la précédente modulo la taille de la liste obtenue. En appliquant

cette méthode on obtient, à la fin de la récursion, la même liste dans les deux cas. Ce qui nous permet de considérer que nous partons d'une liste bitonique simple et non générale avant d'appliquer l'algorithme *minmax* puisque nous allons montrer qu'une liste bitonique simple et générale sont équivalentes après l'application récursive de l'algorithme *minmax*.
Pour prouver cela nous définissons des nouvelles variables :

```
sam,sp,spam : VAR nat
SA,SAp : VAR Arra
```

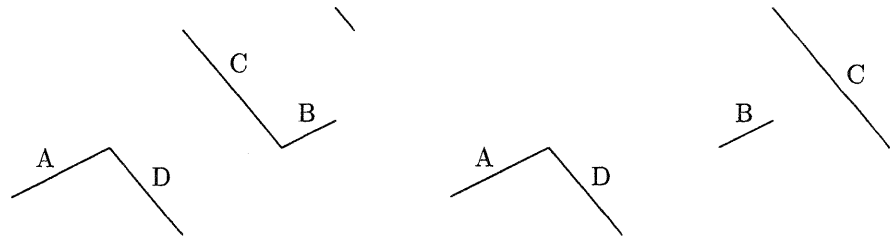
La variable `sam` est la valeur du décalage. Comme on l'ajoute à une variable et qu'on applique un modulo dessus, on l'appelle en anglais « shift after modulo ». `sp` et `spam` sont respectivement les valeurs de `s` et `sam` à l'état suivant. `SA` est un tableau contenant tous les éléments de `A` après le décalage et `SAp` est l'état suivant du tableau.
On définit d'abord quelques théorèmes simples (`R` est une relation sur les naturels) :

```
  R : var PRED[[nat,nat]]

  SHIFT(A,SA,hi,low,nb,nbp,sam,s,R) : bool=
((hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1) and s>=0 and s<nb and
 (FORALL x: x>=low and x<=hi and sam=IF x+s>hi THEN s-nb ELSE s ENDIF and
   A(x)=SA(x+sam)) )
=> (FORALL x:x>=low and x<low+nbp and R(A(x),A(x+nbp))
    => IF x+sam<low+nbp THEN R(SA(x+sam),SA(x+sam+nbp))
       ELSE R(SA(x+sam),SA(x+sam-nbp)) ENDIF)

  SHIFT_0 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,>=)
```

Nous définissons les lemmes `SHIFT_1`, `SHIFT_2` et `SHIFT_3` en remplacant respectivement l'opérateur `>=` par les opérateurs `<=`, `>` et `<`. Puis nous définissons la formule `SHIFT_PROP` comme la synthèse des quatre théorèmes précédents. La formule `SHIFT` est utilisée par les formules `SHIFT_`i, $i \in [0..3]$ qui utilisent les différents opérateurs de comparaisons (`>=`, `<=`, `>` et `<`). Par exemple, le théorème `SHIFT_0` signifie que si `A(x)=SA(x+sam)`, `SA` est une permutation circulaire gauche de `A`, et si tous les éléments dans la première moitié de `A` sont supérieurs ou égaux à tous les éléments de la seconde moitié, alors on connaît la position des éléments dans `SA`, qui dépend de la valeur de `x+sam<low+nbp`.



(a) après *minmax* (b) après le décalage

FIG. 8.9 – *En décalant la seconde liste, on obtient une liste bitonique simple*

Donc, avec ces propriétés, on peut construire les deux théorèmes suivants. Le premier traite la première moitié de la liste alors que le second s'occupe de la seconde moitié. Ces deux théorèmes

sont assez difficiles à prouver car il existe de nombreux cas. La figure 8.9 montre que par le décalage de la seconde moitié de la liste obtenue par l'algorithme *minmax*, on obtient une liste bitonique simple (liste croissante suivie d'une liste décroissante).

```
  SHIFT : LEMMA
(hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and s>=0 and s<nb and
  (FORALL x: x>=low and x<=hi and
    EXISTS sam: sam=IF x+s>hi THEN s-nb ELSE s ENDIF and A(x)=SA(x+sam) ) and
  BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp) and
  BitonicMin(SA,SAp,low,nbp) and BitonicMax(SA,SAp,low,nbp)
) and (FORALL sam: SHIFT_PROP(A,SA,low,nbp,sam) ) =>
       (FORALL sp: sp=IF s>nbp THEN s-nbp ELSE s ENDIF  =>
        (FORALL x,spam:x>=low and x<low+nbp and spam=IF x+sp>=low+nbp THEN
         sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam) ) )
```

On construit également un lemme SHIFT2 de manière similaire à SHIFT. On remplace simplement dans la dernière ligne de `SHIFT => Ap(x)=SAp(x+spam)` par `=> Ap(x+nbp)=SAp(x+nbp+spam)` pour obtenir SHIFT2. La signification de ce lemme SHIFT2 est : si la séquence SA est une permutation circulaire gauche de la séquence A (le décalage étant s) et si on applique l'algorithme *minmax* sur les deux séquences A et SA, alors le nouveau décalage (sp pour s à l'état suivant) sur la seconde liste obtenue est égal à s modulo nbp et Ap est la permutation circulaire gauche de la séquence SAp.
En appliquant récursivement les propriétés sur les séquences, la taille de s décroit et après le dernier pas, sa valeur est zéro.

Remarque 8.3.4 *Maintenant nous donnons un exemple d'une erreur dans une version précédente de notre spécification. Quand nous étions en phase d'élaboration de la propriété SHIFT, la dernière partie d'une version précédente du lemme était :*

```
FORALL x,spam:x>=low and x<=hi and spam = IF x+sp>low+nbp THEN
sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam)
```

à la place de

```
FORALL x,spam:x>=low and x<low+nbp and spam=IF x+sp>=low+nbp THEN
sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam)
```

Il est clair que la version précédente ne pouvait pas être prouvée.
En dérivant le séquent de ce lemme (dans la précédente version), on obtenait, après avoir caché certaines formules :

```
-1    nbp!1 = nb!1 / 2
[-2]    A!1(nbp!1 + x!1) = SA!1(nbp!1 - nb!1 + s!1 + x!1)
  |-------
[1]    A!1(nbp!1 + x!1) = SA!1(s!1 + x!1)
```

En analysant ce résultat on peut voir qu'il faut remplacer `x<=hi` *par* `x<=low+nbp`.

8.3.4 Fin de la preuve

Comme nous avons produit seulement des listes croissantes avec cette spécification, nous prouvons qu'en inversant une liste croissante, nous obtenons une liste décroissante. Ce théorème est facile à prouver :

```
  Reverse(A,Ap,low,hi) : bool =
```

```
(FORALL x:x>=low and x<=hi => Ap(x)=A(hi+low-x))

  INC_TO_DEC_LIST :  LEMMA
(hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
 (FORALL x,y: x>=low and y<=hi and x<=y => A(x)<=A(y)) and Reverse(A,Ap,low,hi) )
=> (FORALL x,y: x>=low and y<=hi and x<=y => Ap(x)>=Ap(y))
```

Une autre propriété fondamentale à prouver est que tous les éléments de la première liste, obtenue avec l'algorithme *minmax*, sont inférieurs à tous les éléments de la seconde liste. Cette propriété, appelée récursivement, produit un tri correct du tableau, car à la fin, le tableau est composé de listes de taille 1, et chaque élément est trivialement plus petit que son successeur.

```
  FirstInfSecond(A,low,hi,nbp) : bool =
FORALL x : x>=low and x<low+nbp => FORALL y : y>=low+nbp and y<=hi => A(x)<=A(y)

  FIRST_INF_SECOND : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
  (FORALL x : x>=j and x<low+nbp => A(x)>=A(j-1)) and
  (FORALL y : y>=low+nbp and y<=j+nbp => A(y)>=A(j+nbp))
) and BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  => FirstInfSecond(Ap,low,hi,nbp)
```

Avec ce théorème on spécifie que, si on a une liste bitonique avec les pivots `i` et `j` convenables, et si on applique l'algorithme *minmax* sur la séquence, alors tous les éléments de `x` de la première liste obtenue sont inférieurs à tous les éléments `y` de la seconde liste.
Maintenant il faut traiter tous les cas spéciaux dont nous ne nous sommes pas encore occupés.

```
  OTHER_CASE : THEOREM
hi>low and nbp>=1 =>(not (EXISTS j:j>low and j<=low+nbp-1 and A(j-1)<=A(j-1+nbp)
and A(j)>=A(j+nbp) )
 <=> (FORALL j: j<=low or j>low+nbp-1 or A(j-1)>A(j-1+nbp) or A(j)<A(j+nbp)) )
```

Dans la plupart des preuves précédentes, on considère que l'on est dans l'état où `j>low and j<=low+nbp-1`, c'est-à-dire que `j` existe, donc si cette condition n'est pas satisfaite, cela signifie que `A(j-1)>A(j-1+nbp)` ou `A(j)<A(j+nbp)`. Nous obtenons cette propriété avec le théorème `OTHER_CASE`.
Donc, avec ces deux cas (`A(j-1)>A(j-1+nbp)` ou `A(j)<A(j+nbp)`), nous devons prouver les propriétés similaires à celles que nous avons vues dans le cas normal (où `j>low et j<=low+nbp-1`), c'est-à-dire que nous devons prouver qu'en appliquant l'algorithme *minmax* nous obtenons des listes bitoniques dans lesquelles tous les éléments de la première liste obtenue sont inférieurs à tous les éléments de la seconde liste obtenue.
Le théorème `PROP_NOJ_0` est utilisé pour conclure, que si on a une liste bitonique et si tous les éléments indexés par `x` dans la première moitié de la liste sont supérieurs à tous les éléments indexés par `x+nbp` dans la seconde moitié, il en découle que pour tout `x` et `y`, index de la seconde moitié, tel que `x` soit inférieur ou égal à `y` alors nous avons `A(x)` supérieur ou égal à `A(y)`. Nous utilisons cette propriété, dans le théorème `INF_NOJ`, afin de prouver que dans ce cas, nous obtenons deux listes bitoniques avec lesquelles nous avons la propriété d'infériorité (théorème `INF_NOJ`). Ces théorèmes sont dans l'annexe C. Nous construisons également des théorèmes similaires `PROP_NOJ_1`, `BIT_NOJ_1` et `INF_NOJ_1` afin de traiter le cas où tous les éléments `x` de

la première moitié de la liste sont inférieurs à tous les éléments x+nbp de la seconde moitié de la liste.
Et finalement, nous devons prouver les mêmes théorèmes quand nous avons respectivement seulement une liste croissante (théorèmes BIT_INC et INF_INC) et seulement une liste décroissante (théorèmes BIT_DEC et INF_DEC).

```
  BIT_INC : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 =>
IncreasingList(A,low,hi) and BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  => IncreasingList(Ap,low,low+nbp-1) and IncreasingList(Ap,low+nbp,hi)

  INF_INC : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 =>
IncreasingList(A,low,hi) and BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp)
  => FirstInfSecond(Ap,low,hi,nbp)
```

Pour obtenir les théorèmes BIT_DEC et INF_DEC, on a juste à remplacer IncreasingList par DecreasingList.
Maintenant nous pouvons résumer la preuve que nous avons faite.

- Ayant une liste bitonique simple de 2^p éléments et appliquant notre algorithme, on obtient deux listes bitoniques de taille 2^{p-1} (une liste bitonique simple et une permutation circulaire gauche d'une liste bitonique simple). Plus précisément, le théorème BIT_0 prouve que la première liste obtenue après l'algorithme est une liste bitonique simple. Le théorème BIT_1 prouve que la seconde liste obtenue est une liste bitonique générale (permutation circulaire gauche d'une liste bitonique). Dans ces deux preuves, nous utilisons une liste bitonique simple avant l'algorithme puisque nous avons vu qu'une liste bitonique générale est équivalente à une liste bitonique simple après avoir appliqué récursivement l'algorithme *minmax*. Pour les listes croissantes et décroissantes, nous utilisons les théorèmes BIT_INC et BIT_DEC. Lorsque tous les éléments de la liste croissante d'une liste bitonique simple sont supérieurs (inférieurs) à tous les éléments de la liste décroissante, nous utilisons le théorème BIT_NOJ_0 (BIT_NOJ_1), c'est-à-dire lorsqu'on ne peut pas définir le pivot j.
- Tous les éléments dans la première liste bitonique obtenue sont inférieurs à tous les éléments de la seconde liste. Nous montrons cette propriété pour une liste simple avec le théorème FIRST_INF_SECOND. Lorsqu'on ne peut pas définir le pivot j, on utilise les théorèmes INF_NOJ_0 et INF_NOJ_1. Lorsque la liste est seulement croissante ou seulement décroissante, nous utilisons les théorèmes INF_INC et INF_DEC.
- Une permutation circulaire gauche d'une liste bitonique est équivalente, à la fin de la récursion de l'algorithme *minmax*, à une liste bitonique. Nous montrons cette propriété avec les deux théorèmes SHIFT et SHIFT2.

Donc, partant avec des listes de taille un, et construisant à chaque étape des séquences de taille deux fois plus grosses (alternativement croissante et décroissante), nous obtenons à la fin de l'algorithme une liste croissante de taille 2^p. Par conséquent, à l'aide de tous ces théorèmes, nous pouvons être sûr que la spécification suivante est correcte.

8.4 Spécification de l'algorithme de tri bitonique en PVS

Maintenant que nous avons prouvé toutes les propriétés que la spécification devait satisfaire, nous pouvons construire une spécification de l'algorithme de tri. Pour cela, nous devons examiner comment cet algorithme s'exécute. Il commence par créer alternativement des listes croissantes

et décroissantes de taille 2. Puis, à chaque étape p, il construit alternativement des listes de taille 2^p. Chaque étape p est composée de p pas. Donc nous avons besoin d'avoir une fonction récursive pour concaténer, à chaque pas p, une liste croissante et une liste décroissante de taille 2^{p-1}, afin de construire soit une liste croissante soit une liste décroissante. Mais nous avons vu que l'algorithme nécessite p pas pour effectuer cette tâche. Donc la fonction de concaténation a besoin d'appeler cette fonction qui découpe récursivement une liste de 2^p en deux listes de 2^{p-1}. Ces fonctions sont respectivement appelées `BitonicMerge` et `BitonicSplit`.
La fonction `BitonicMerge` est appelée au début de l'algorithme avec la valeur `n_cur` $= 2$. Cette valeur est la taille des premières listes construites par l'algorithme. Ensuite, à chaque récursion, la valeur de `n_cur` est multipliée par 2, jusqu'à ce qu'elle atteigne la valeur de `nb`. Alternativement, on construit une liste croissante et une liste décroissante en utilisant la fonction `Reverse`.
La fonction `BitonicSplit` sépare simplement une liste en deux en utilisant les fonctions `BitonicMin` et `BitonicMax`.

```
bitonic_sort_algo  : THEORY
  BEGIN

  Arra : TYPE = [nat->nat]
  n,x,y,nb,nbp,n_cur,n_curp,low,hi,b,m:VAR nat
  A,Ap,App,Appp : VAR Arra

  BitonicMin(A,Ap,low,nbp) : bool =
    FORALL x: x>=low and x<low+nbp => Ap(x) = min(A(x),A(x+nbp))

  BitonicMax(A,Ap,low,nbp) : bool =
    FORALL x:  x>=low and x<low+nbp => Ap(x+nbp) = max(A(x),A(x+nbp))

  BitonicSplit(A,App,low,hi,nb) : RECURSIVE bool =
    IF nb=1 THEN A=App
    ELSE
      EXISTS Ap,nbp : nbp=nb/2 and BitonicMin(A,Ap,low,nbp) and
      BitonicMax(A,Ap,low,nbp) and
      BitonicSplit(Ap,App,low,hi-nbp,nbp) and BitonicSplit(Ap,App,low+nbp,hi,nbp)
    ENDIF
  MEASURE nb

  Reverse(A,Ap,low,hi) : bool = (FORALL x:x>=low and x<=hi => Ap(x)=A(hi+low-x))

  BitonicMerge(A,Appp,low,hi,nb,n_cur) : RECURSIVE bool =
    IF n_cur>nb THEN A=Appp ELSE
      EXISTS Ap,App,n_curp:
       (FORALL b : b>=0 and b<nb/n_cur IMPLIES
         BitonicSplit(A,Ap,low+b*n_cur,low+(b+1)*n_cur-1,n_cur) and
         IF EXISTS m: b=2*m+1 THEN
           App=Ap ELSE Reverse(Ap,App,low+b*n_cur,low+(b+1)*n_cur-1)
         ENDIF
       ) and n_curp=n_cur*2 and BitonicMerge(App,Appp,low,hi,nb,n_curp)
    ENDIF
  MEASURE nb-n_cur

  END bitonic_sort_algo
```

Si par exemple, nous voulions trier le tableau de la figure 8.2, nous devrions appeler la fonction

suivante `BitonicSort(A,Ap,1,16,16,2)` où `A` contient initialement les nombres non triés.

8.5 Comment paralléliser le tri bitonique?

Pour que cet algorithme trie en parallèle un tableau de nombres, nous devons préciser quelle parties de l'algorithme peuvent être exécutées en parallèle. En analysant la spécification de ce tri, on peut constater qu'à chaque appel de fonction `BitonicMerge`, on appelle un certain nombre de fois (`nb/n_cur` pour être exact) la fonction `BitonicSplit`. Comme chaque appel de fonction travaille sur une partie différente du tableau, il suffit, pour paralléliser cet algorithme d'appeler en parallèle toutes les fonctions `BitonicSplit`. Notons qu'une fois sur deux, nous appelons également la fonction `Reverse` pour obtenir une liste décroissante à partir d'une liste croissante, donc cette fonction doit également être exécutée en parallèle.
En reprenant la spécification précédente, il suffit d'exécuter toutes les fonctions appelées par le `FORALL` de la fonction `BitonicMerge` en parallèle.
Nous avons écrit en C++, cet algorithme, en annexe D nous donnons le programme entier. Voici la directive de compilation que nous avons ajoutée afin de paralléliser le programme.

```
void bitonicmerge(float* a, long low,long hi,long n_cur,long n_stop)
{
  if(n_cur<=n_stop)
  {
    long b;
    #pragma parallel local(b)
    {
    #pragma pfor
    for(b=0; b<(hi-low+1)/n_cur; b++)
      {
        if (b%2==0)
          bitonicsplitinc(a,low+b*n_cur,low+(b+1)*n_cur-1,n_cur);
        else
          bitonicsplitdec(a,low+b*n_cur,low+(b+1)*n_cur-1,n_cur);
      }
    }
    n_cur=n_cur*2;
    bitonicmerge(a,low,hi,n_cur,n_stop);
  }
}
```


Dans cette fonction, on indique au compilateur que toutes les itérations de la boucle `for` doivent être exécutées en parallèle. Juste avant cette boucle, on spécifie qu'une seule variable est locale à la boucle. Cette notation est équivalente au `PRIVATE` utilisé avec OpenMP.

8.6 Travaux relatifs

Les travaux relatifs au tri bitonique fournissent différentes approches afin de prouver la correction de cet algorithme. Batcher dans [Batcher, 1968] donne une ébauche de la preuve de la règle itérative pour les tris bitoniques, tous les cas décrits précédemment ne sont pas traités. Gamboa dans [Gamboa, 1997] utilise les Powerlists, définies par Misra dans [Misra, 1994], pour prouver que le tri bitonique est équivalent au tri de Batcher. Bien que la preuve ne soit pas directe, puisqu'elle utilise la preuve de Batcher, elle est construite en utilisant le prouveur de théorème

ACL2. Bilardi et Nicolau dans [Bilardi et Nicolau, 1986] donnent des propriétés relatives aux séquences bitoniques, mais elles utilisent des propriétés prouvées par d'autres personnes ; par exemple, elles utilisent le fait qu'une sous-séquence d'une liste bitonique est bitonique.

8.7 Conclusion

Nous avons montré que le tri bitonique est correct en utilisant des connaissances simples sur cet algorithme. En fait, nous avons commencé par étudier ce tri bien connu et nous avons décrit (de manière formelle) ses principales propriétés. Ensuite nous avons prouvé que ces propriétés étaient correctes : PVS a été une grande aide à cet effet car son langage de spécification permet d'écrire des propriétés très proches de ce que l'on souhaite exprimer. Dans cet exemple, nous avons souvent pu exprimer directement ce que nous voulions spécifier avec PVS.
La plupart des propriétés utilisées pendant la preuve n'ont pas été identifiées avant le début de la preuve. En fait, nous avons formulé des versions provisoires, et c'est seulement après la preuve que l'on s'est rendu compte de la validité de ces définitions. Le développement de la preuve de cet algorithme en parallèle avec sa spécification est une nouvelle approche, puissante, pour le développement de logiciel.
La parallélisation du programme séquentiel est trivial sur une machine à mémoire partagée car toutes les itérations de la boucle `for` de la fonction `bitonicmerge` sont exécutables en parallèle par définition du tri bitonique. En effet, on constitue des listes bitonique de taille croissante sur des parties différentes du tableau.
Ce travail illustre le fait que les propriétés de programmes sont prouvables à l'aide d'un prouveur de théorèmes interactif. La difficulté était de décrire au mieux les propriétés intermédiaires nécessaires pour réaliser la preuve. Dans le cas du tri bitonique, certaines propriétés sont non triviales et PVS nous a permis de le faire. Pour le chapitre suivant qui consiste à définir une méthode permettant de montrer qu'une parallélisation basée sur une décomposition de domaine, ce travail nous a permis de comprendre comment fonctionne PVS afin de savoir ce qu'il est capable de prouver. La connaissance de l'outil permet de réaliser l'abstraction adéquate avec ce qu'on souhaite prouver.

9

Méthode pour prouver la correction d'une parallélisation

9.1 Méthode pour la parallélisation

La correction d'applications parallèles en « grandeur nature » est un problème difficile à aborder. Actuellement, il est possible de faire la preuve d'algorithmes parallèles mais il n'existe pas vraiment de méthodes reconnues par tout le monde pour les applications. Dans le chapitre 3, nous avons présenté deux approches, celle de Massingill [Massingill, 1998] et de Sere [Sere, 1990], qui ont été utilisées pour constuire des programmes parallèles corrects et une approche pour vérifier qu'un algorithme parallèle est correct.
Pour les parallélisations utilisant l'archétype de décomposition de domaines, nous avons développé une méthode générale pour montrer qu'une parallélisation est correcte. Nous utilisons un environnement de développement de preuve, ce qui constitue une nouveauté. Sa différence principale avec les approches d'Owicki-Gries [Owicki et Gries, 1976], de Massingill et de Sere, réside dans le fait que notre approche permet de prouver que la parallélisation d'un programme séquentiel est correcte. Néanmoins notre méthode permet de paralléliser un programme tout en prouvant que la parallélisation est correcte. Cette méthode repose sur le raisonnement à partir des post-conditions que l'on établit en réalisant une abstraction de la version séquentielle et de la version parallèle de notre programme.

9.1.1 Méthode pour montrer qu'une parallélisation est correcte

Lorsque l'on parallélise un algorithme ou un code séquentiel, nous avons toujours comme objectif que la parallélisation soit correcte, afin que la version parallèle et la version séquentielle du programme donnent les mêmes résultats. Nous avons toujours un autre objectif qui consiste à obtenir des résultats plus rapidement ou de pouvoir traiter plus de données dans le même temps.

Contexte

Pour prouver qu'un programme, une fois parallélisé, donne le même résultat qu'en séquentiel, il faut pouvoir apporter la preuve qu'à un ordonnancement près, les calculs sont identiques. Comme il existe de nombreux moyens de paralléliser une application, il y a par conséquent des techniques spécifiques à ces moyens. Ainsi, prouver qu'une boucle, une fois parallélisée, effectue le même calcul que la boucle séquentielle dont elle est issue, est différent de prouver qu'un programme décomposé en plusieurs sous-programmes donne le même résultat que le programme initial. Dans

le premier cas, il faut se placer au niveau des itérations de la boucle pour analyser finement les dépendances, alors que dans le second cas, il faut bien sûr analyser les dépendances, mais cette fois-ci au niveau du programme tout entier. Donc suivant ce que l'on souhaite prouver, on adapte la preuve afin qu'elle capture le niveau d'analyse choisi, c'est-à-dire les itérations d'une boucle ou un programme.
Notre démarche se place dans le cas où nous effectuons une décomposition de domaines, c'est-à-dire que nous divisons le domaine sur lequel nous travaillons initialement en plusieurs sous-domaines. Avec une perspective de décomposition de domaines, nous appliquons sensiblement le même algorithme sur chaque sous-domaine et nous effectuons ensuite les calculs concernant les données qui n'ont pas été traitées. De nombreuses parallélisations sont entreprises de cette manière car ce type de parallélisation permet d'utiliser sensiblement le même code du programme séquentiel sur chaque sous-domaine du programme parallèle. Cependant pour certains programmes, les nombreuses interactions entre les composants du calcul rendent le travail de décomposition très complexe.
Une parallélisation par décomposition de domaines permet d'obtenir des solutions locales que l'on va ensuite propager afin d'avoir une solution globale. Ce passage des solutions locales à une solution générale est souvent la cause d'erreurs dans les programmes parallèles.

Choix des post-conditions

Afin de prouver qu'une parallélisation par décomposition de domaines est correcte, nous allons raisonner sur l'algorithme séquentiel et, par abstraction, trouver une post-condition de cet algorithme, c'est-à-dire une condition qui est vraie après l'exécution de l'algorithme. Pour définir cette post-condition, il faut « regarder » le code ou l'algorithme et « voir » ce qu'il fait sur les données. Il n'y a pas, malheureusement, de moyen de trouver automatiquement la « bonne » post-condition qui va être utile pour la preuve. Actuellement il existe des outils pour trouver une post-condition d'un algorithme mais pour ce problème, ce qui nous intéresse, c'est de trouver celle qui nous permettra de faire la preuve. Ce critère est subjectif et ne dépend que de la connaissance de l'utilisateur dans ce domaine et de son intuition. Suivant la post-condition que l'on choisit, il faudra effectuer des preuves très longues et difficiles si on précise trop le contexte, ou au contraire, on ne pourra rien prouver car la post-condition ne nous sera d'aucune utilité en raison de sa faiblesse.
Ensuite, ayant défini la post-condition du programme séquentiel, il faut définir la post-condition de chaque partie parallèle. En général, on applique le même algorithme sur chaque partie parallèle, donc on définit une seule post-condition, qui dépend du domaine, pour le cas parallèle.
Finalement, il nous faut définir la post-condition du code qui va réaliser le collage entre les différentes parties parallèles. Ce collage a pour but d'homogénéiser les différents résultats partiels obtenus en parallèle, comme représenté sur la figure 4.1 de la page 50 du chapitre 4. On définit la post-condition de ce que nous appelons dans la suite : « la colle », de la même manière que nous avons défini la post-condition du programme séquentiel et du programme parallèle.
Comme il est difficile de donner une méthode générale pour réaliser l'abstraction du programme séquentiel, de la colle et des parties parallèles, cependant voici des critères importants à respecter :

- L'abstraction du programme séquentiel, de la colle et des parties parallèles doit obligatoirement faire apparaitre les données qui contiennent le résultat du programme et les données qui sont distribuées. Nous qualifions ces données de cruciales pour le calcul. En effet, les données qui sont distribuées vont nous permettre de vérifier que les calculs sont distribués correctement et les données contenant un résultat vont nous permettre de vérifier que le

résultat du programme est correct. Par exemple, l'abstraction d'un programme qui résoud une équation de Poisson doit faire apparaître la grille comme données cruciales car la grille est distribuée entre les différents processeurs dans la version parallèle du programme et car elle contient un résultat. Un programme qui modélise un problème des N-corps doit différencier les interactions entre les corps proches les uns des autres, des interactions entre des corps éloignés.

- L'abstraction du programme séquentiel doit modéliser une partie du calcul entre toutes les données. Suivant la complexité du programme étudié, on abstrait ou non une bonne partie des calculs afin de conserver uniquement l'essentiel du calcul. En reprenant le problème des N-corps, si la modélisation fait intervenir des calculs entre deux corps, l'abstraction doit modéliser que deux corps interagissent sans pour autant modéliser que le calcul utilise une racine carrée ou une racine cubique de la vitesse.
- L'abstraction des parties parallèles doit si possible être similaire à l'abstraction du programme séquentiel. Cela rend la preuve de parallélisation plus facile. En reprenant l'abstraction du programme séquentiel, l'abstraction des parties parallèles doit faire apparaître que certains calculs sont effectués localement par un processeur, sans échanger des données, alors que d'autres calculs sont réalisés avec des données disposées sur différents processeurs. La différentiation entre les calculs utilisant des données locales à un processeur et les calculs utilisant des données distantes à un processeur est primordiale.
- Finalement l'abstraction de la colle doit modéliser le fait que les calculs faisant intervenir des données disposées sur plusieurs processeurs doivent être effectués. Avec l'exemple des N-corps, l'abstraction de la colle doit modéliser les interactions entre les corps éloignés comme l'abstraction du programme séquentiel, c'est-à-dire en modélisant grossièrement l'interaction.

Preuve

Maintenant que nous avons défini les post-conditions du programme séquentiel, du programme parallèle et de la colle, nous pouvons essayer de prouver que notre parallélisation est correcte. Pour cela, nous devons prouver que les post-conditions des parties parallèles plus la post-condition de la colle impliquent la post-condition du programme séquentiel.
Formalisons cette règle :

- Soit PS la partie séquentielle (cette partie peut être le programme en entier) que nous avons parallélisée par l'intermédiaire de n parties parallèles, notées PP_i, $i \in [1 \dots n]$. Soit C la partie qui harmonise les données que nous appelons également la colle. Notons $post(P)$ la post-condition d'une partie de code P. Alors il faut prouver la condition suivante :

$$\bigwedge_{i=1}^{n} post(PP_i) \wedge post(C) \Rightarrow post(PS) \tag{9.1}$$

Pour effectuer cette preuve nous allons utiliser un prouveur de théorèmes dans lequel nous pouvons modéliser les post-conditions que nous avons définies. Afin d'arriver au bout de la preuve, il y a plusieurs conditions à remplir :

- Il faut avoir défini correctement toutes les post-conditions, nous allons voir comment nous convaincre que les post-conditions sont correctes.
- Il faut maîtriser un prouveur de théorèmes, car il est fréquent que le prouveur détecte des cas que l'humain ne considère pas par évidence ou par mégarde.

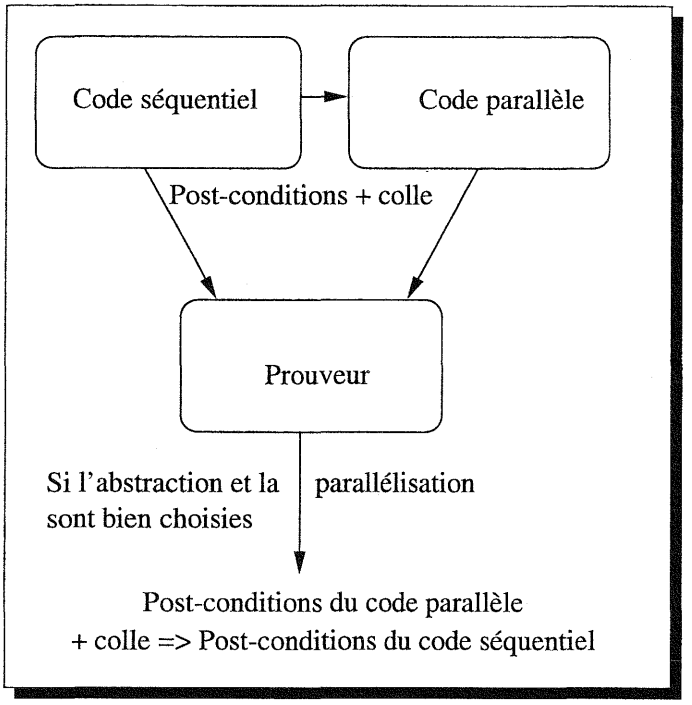


FIG. 9.1 – *preuve qu'une parallélisation est correcte à l'aide d'un prouveur*

La figure 9.1 résume le principe de notre méthode que nous récapitulons :

1. Nous partons du programme séquentiel.
2. Nous construisons le programme parallèle utilisant une décomposition de domaines.
3. Nous élaborons par abtraction les post-conditions du programme séquentiel, des parties parallèles et de la colle.
4. Nous vérifions la validité de ces post-conditions en les traduisant dans le langage du programme source et en vérifiant à partir de plusieurs exécution leur validité (page 100, nous donnons un exemple de traduction). Il est clair que nous n'effectuons pas de preuve, car prouver ces post-conditions revient à prouver le programme. Or cette tâche est trop compliquée, et c'est pour cela que nous proposons notre méthode.
5. Nous utilisons un environnement de développement de preuves pour essayer de prouver la condition 9.1. Cette formule est prouvable si l'abstraction est bien choisie et si la parallélisation est correcte.

Dans la section suivante, nous appliquons notre démarche sur la première simulation de transition de phase que nous avons parallélisée. Ainsi, on comprend mieux comment fonctionne notre méthode. Mais auparavant, nous expliquons la démarche, similaire à celle que nous venons d'exposer, pour paralléliser un programme.

9.1.2 Méthode pour paralléliser à l'aide d'un prouveur

La première méthode proposée nous permet de raisonner avec les post-conditions afin de prouver qu'une parallélisation est correcte. Pour paralléliser une application toujours dans le même

contexte, nous allons utiliser la même technique que précédemment, sans définir la post-condition de la colle. Il faut, comme précédemment, avoir spécifié correctement les post-conditions du programme séquentiel et du programme parallèle. Nous partons du programme séquentiel pour définir ces post-conditions. Les post-conditions du programme parallèle que nous souhaitons développer sont définies en considérant que notre programme utilise une décomposition de domaines.
Ensuite nous allons essayer de prouver que les post-conditions des parties parallèles impliquent la post-condition du programme séquentiel.
En reprennant le formalisme introduit dans la section précédent, nous devons prouver la condition suivante :

$$\bigwedge_{i=1}^{n} post(PP_i) \Rightarrow post(PS) \tag{9.2}$$

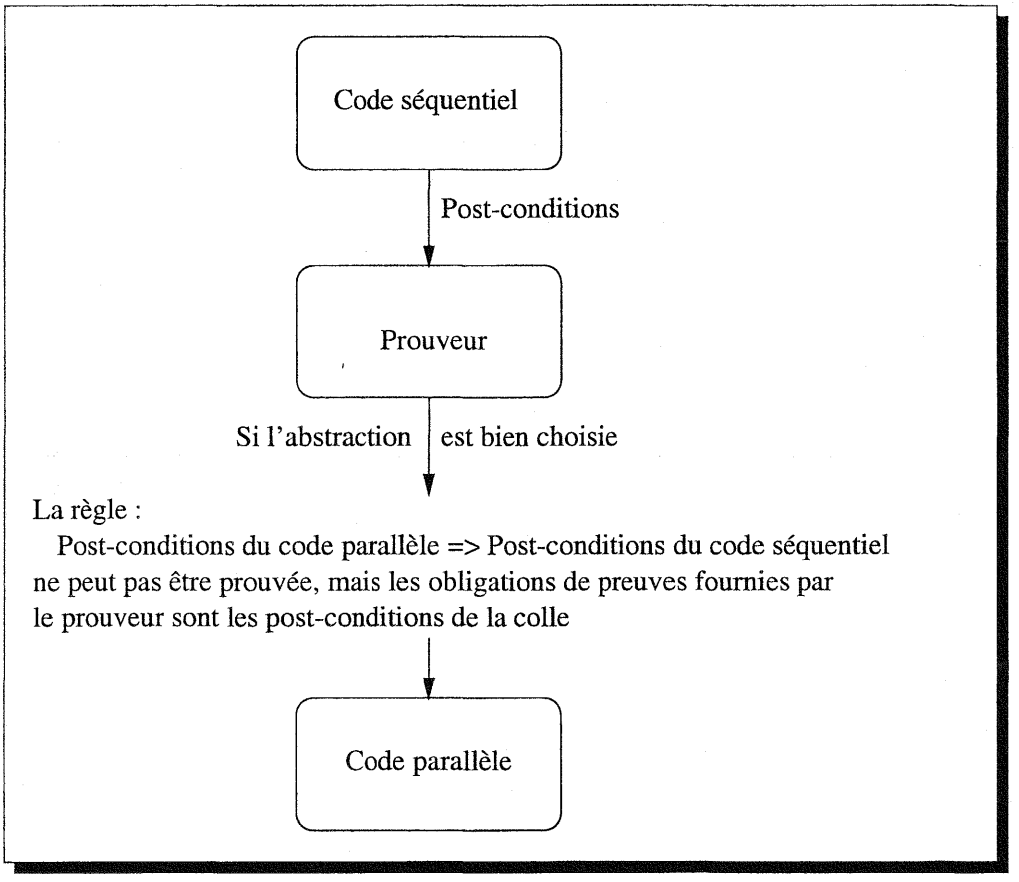


FIG. 9.2 – *parallélisation à l'aide d'un prouveur*

Bien évidemment, nous n'allons pas réussir à faire cette preuve puisque nous n'avons pas spécifié comment unifier les différents résultats obtenus en parallèle. Pour chaque domaine, nous pouvons prouver que les données qui dépendent uniquement des autres données du même domaine, sont identiques à celles du programme séquentiel. Par contre, pour toutes les frontières entre les différents domaines, là où nous n'avons pas pu effectuer les calculs en raison des données qui n'étaient pas présentes, la preuve n'est pas possible. Les obligations de preuves sont en fait les conditions que nous devons remplir afin que les résultats obtenus en parallèle impliquent les

résultats obtenus en séquentiel. Elles ont la forme suivante :

- Pour toutes les données dont les calculs nécessitent des données d'autres domaines (en général ces données sont proches des frontières entre les domaines), il faut prouver que les propriétés sur les données obtenues en parallèle impliquent des propriétés équivalentes sur les données obtenues en séquentiel.

Ces propriétés sont exploitables pour générer les post-conditions de la colle. Il suffit, suivant le prouveur que l'on utilise, de retranscrire les obligations de preuve dans le langage de spécifications. Finalement, lorsque nous possédons toutes les informations concernant la post-condition de la colle, nous pouvons refaire la preuve en tenant compte de la colle. Si tout est correct, nous devons pouvoir prouver la condition 9.1.
Ainsi, nous avons parallélisé un code à l'aide d'un prouveur de théorèmes et nous avons réalisé la preuve que la parallélisation est correcte par la même occasion. La figure 9.2 résume la démarche à suivre.
Récapitulons la démarche à suivre :

1. Nous partons du programme séquentiel.
2. Nous réalisons une abstraction du programme séquentiel, puis nous réalisons les post-conditions des différentes parties parallèles de notre futur programme parallèle que nous voulons développer en utilisant une décomposition de domaine.
3. Nous vérifions la post-condition du programme séquentiel en traduisant celle-ci dans le langage du programme source et en vérifiant à l'exécution sa validité sur plusieurs exécution.
4. Nous utilisons un environnenent de développement de preuves pour essayer de prouver la condition 9.2. Certaines obligations de preuves fournies par le prouver pour terminer cette preuve ne sont pas prouvables, mais si notre abstraction est correcte, ces obligations de preuves constituent la post-condition de la colle.
5. Nous pouvons construire la post-condition de la colle et prouver la condition 9.2, ce qui nous permet de construire notre programme parallèle.

Maintenant que nous avons défini notre méthode, nous pouvons regarder comment la mettre en pratique sur la première simulation de transition de phase que nous avons réalisée.

9.2 Preuve de la correction de la « parallélisation des graphes » du premier système de transition de phase

Il s'agit de prouver la parallélisation présentée dans le chapitre 5. Nous utilisons PVS [Crow *et al.*, 1995; Owre *et al.*, 1998], comme prouveur de théorèmes, car nous le maîtrisons bien, et qu'il est, à notre avis, le plus puissant pour effectuer le travail que l'on souhaite faire.
Nous considérons le cas le plus général possible où nous disposons de `n` processeurs en largeur et `m` processeurs en hauteur. Nous réalisons une abstraction des données du programme. Dans le code Fortran, nous avons deux tableaux pour représenter les liens horizontaux `HLink` et verticaux `VLink`, donc nous allons les représenter de manière équivalente dans PVS. Après l'étape de création des liens, les graphes sont dans un tableau nommé `Spin`.
Dans PVS, pour représenter ces variables, nous utilisons deux types de tableaux (`bool` et `nat`). Nous utilisons la notion de variables *primées* pour représenter l'état d'une variable à l'état suivant de l'état courant, comme dans TLA [Lamport, 1994] en ajoutant la lettre « p » après une variable. Ainsi, la variable `Spinp` contient tous les graphes partiels de la grille obtenus en séquentiel, et la variable `Spin2p` représente la grille après le calcul des graphes en parallèle.

```
Array2_nat : TYPE = [nat,nat->nat]
```

```
Array2_bool : TYPE = [nat,nat->bool]
Spinp,Spin2p : Var Array2_nat
HLink,VLink : Var Array2_bool
```

Maintenant nous devons définir la post-condition obtenue après l'exécution de l'algorithme séquentiel. Comme nous utilisons le même algorithme en parallèle, cette post-condition doit être également vérifiée par chaque processeur. Les variables `off_x` et `off_y` représentent respectivement un *offset* en `x` et en `y` afin de pouvoir appliquer le même algorithme à des endroits différents de la grille. Chaque processeur l'applique à un endroit qui lui est propre.

```
Algo_seq(HLink,VLink,off_x,off_y,width,height,Spinp) : bool =
(FORALL x,y: x>=off_x and x<off_x+width-1 and y>=off_y and
y<=off_y+height-1 =>
    (HLink(x,y) =>  Spinp(x,y)=Spinp(x+1,y)))
and
(FORALL x,y: x>=off_x and x<=off_x+width-1 and y>=off_y and
y<off_y+height-1 =>
    (VLink(x,y) =>  Spinp(x,y)=Spinp(x,y+1)))
```

Cette formule est une abstraction de la partie du programme qui crée les graphes (en fait, dans notre cas, c'est une post-condition). On différencie les liens horizontaux (première partie) des liens verticaux. Pour les liens horizontaux, on spécifie que pour tout `x` et `y` appartenant au domaine de l'algorithme, si après l'exécution de celui-ci, on a un lien horizontal en `(x,y)`, alors `Spinp(x,y) = Spinp(x+1,y)`.
Ayant spécifié la post-condition de l'algorithme, il nous faut définir la condition de la « colle ». Prenons `VerticalGlueing` par exemple, on a : pour tout `x` et `y` appartenant au domaine, alors en présence d'un lien vertical, les valeurs de `Spin2p(x,y)` et de `Spin2p(x+1,y)` sont égales.

```
VerticalGlueing(HLink,VLink,off_x,width,height,Spin2p) : bool =
   FORALL x,y: (x=off_x and y>=0 and y<height-1 =>
    (HLink(x,y) => Spin2p(x,y)=Spin2p(x+1,y)))

HorizontalGlueing(HLink,VLink,off_y,width,height,Spin2p) : bool =
   FORALL x,y: (y=off_y and x>=0 and x<width-1 =>
    (VLink(x,y) => Spin2p(x,y)=Spin2p(x,y+1)))
```

Maintenant nous devons définir exactement ce que nous voulons prouver. On désire prouver que si on applique l'algorithme parallèle, on obtient bien le même résultat que si on applique l'algorithme séquentiel en tous points de la grille.
Les variables `n` et `m` représentent respectivement le nombre de processeurs en largeur et en hauteur (donc on a `n*m` processeurs). La largeur et la hauteur d'une sous-grille pour un processeur dans le cas parallèle sont respectivement `band_w` et `band_h`. Dès lors que ces variables ont des valeurs valides, on applique l'algorithme séquentiel et la grille dans ce cas est représentée par `Spinp`. Dans le cas parallèle, la grille est représentée par `Spin2p`. On applique aussi l'algorithme séquentiel sur chaque sous-grille ; `u1*band_w` et `v1*band_h` représentent les *offsets* de la sous-grille. On applique également `n-1` fois `VerticalGlueing` et `m-1` fois `HorizontalGlueing` pour coller les sous-grilles. Puis ayant appliqué les deux algorithmes séquentiel et parallèle, on souhaite montrer (après le `=>` seul sur une ligne) que pour tout `x` et `y` appartenant à la grille complète, si en `(x,y)`, la valeur séquentielle est la même que la valeur parallèle (`Spinp=Spin2p`) et s'il y a un lien horizontal, alors en `(x+1,y)` les valeurs des grilles en séquentiel et en parallèle sont identiques et égales à celles en `(x,y)`. On obtient bien sûr une condition symétrique pour `VLink`.

```
EQUI : THEOREM FORALL (m,n,band_w,band_h:nat) :
```

```
(n>=1 and m>=1 and band_w*n=width and band_h*m=height and band_w>=1 and band_h>=1) =>
(( Algo_seq(HLink,VLink,0,0,width,height,Spinp) and
 (FORALL (u1,v1:nat) : (u1>=0 and u1<n and v1>=0 and v1<m) =>
  (Algo_seq(HLink,VLink,u1*band_w,v1*band_h,band_w,band_h,Spin2p) and
   (u1<n-1 => VerticalGlueing(HLink,VLink,u1*band_w+band_w-1,width,height,Spin2p)) and
   (v1<m-1 => HorizontalGlueing(HLink,VLink,v1*band_h+band_h-1,width,height,Spin2p))
  )))
 =>
    (FORALL x,y: x>=0 and y>=0 and x<width-1 and y<height-1 =>
     (Spinp(x,y)=Spin2p(x,y) and HLink(x,y))
      =>
     (Spinp(x,y)=Spinp(x+1,y) and Spinp(x+1,y)=Spin2p(x+1,y)))
   and   Conditions équivalentes avec VLink(x,y)
)
```

La preuve de ce théorème est relativement simple. Nous donnons schématiquement le raisonnement utilisé. Tout d'abord, nous instancions les variables `u1` et `v1` du second `FORALL` par `floor(x/band_w)` et `floor(y/band_h)`. En introduisant des parties entières, nous devons prouver des propriétés sur celle-ci. Nous avons défini 14 lemmes à ce sujet. Ensuite il faut différencier le cas horizontal du cas vertical[16]. Puis il faut différencier le cas des frontières de l'autre cas. Lorsqu'on est sur une frontière on utilise la formule de la « colle » pour prouver l'obligation de preuve. En tout, nous avons utilisé une centaine de commandes pour prouver ce théorème[17].
On peut se poser une question intéressante à ce niveau : comment vérifier que l'abstraction que nous avons faite correspond au code dont nous sommes parti. Si nous voulons le prouver, il faut évidemment utiliser des méthodes de preuves connues, et nous serons obligés de prouver l'algorithme. Nous voulons éviter de faire cela, car ce qui nous intéresse, c'est de prouver que la parallélisation est correcte. Nous ne souhaitons pas prouver que le programme est correct. Néanmoins nous pouvons tout de même vérifier que notre spécification est valide en injectant toutes les formules PVS dans le code Fortran. Ainsi, à l'exécution du code, si une condition n'est pas vérifiée, on s'en aperçoit. Bien sûr cette démarche n'est pas aussi rigoureuse qu'une preuve, mais elle donne quand même une bonne « intuition » sur la correction de notre abstraction. De plus, si nous avions fait une abstraction trop forte, nous n'aurions rien pu prouver d'intéressant dans PVS. Donc l'injection des formules dans le code nous paraît être la méthode la plus appropriée pour nous convaincre de la pertinence de la spécification [Couturier et Méry, 1998a].
À titre d'exemple, voici le code que nous avons introduit dans le programme séquentiel Fortran après la partie de création des graphes. Ce code permet de vérifier que la formule `Algo_seq` est une post-condition du code de création des graphes.

```
Do u=0,width-2
  Do v=0,height-2
    If (Hlink(u,v).eq.1) then
      If (Spin(u,v).ne.Spin(u+1,v)) Then
        Write (*,*) 'Probleme en ',u,v,'pour le lien horizontal'
      Endif
    Endif
    If (Vlink(u,v).eq.1) then
      If (Spin(u,v).ne.Spin(u,v+1)) Then
        Write (*,*) 'Probleme en ',u,v,'pour le lien vertical'
      Endif
```

16. Mais la preuve est similaire.
17. Nous n'avons pas cherché à réduire ce nombre, mais il est souvent possible de le faire.

```
      Endif
    Enddo
  Enddo
```

9.3 Recherche de la post-condition de la colle en utilisant PVS

Nous avons vu qu'un point important de notre méthode consiste à prouver qu'un programme parallèle est correct par rapport au programme séquentiel dont nous sommes parti. Mais cette méthode nous fournit également un bon moyen de trouver la post-condition de la colle. Pour cela, on définit la post-condition de l'algorithme séquentiel. Ensuite, on partitionne les données de notre programme, en définissant des sous-domaines qui correspondent aux sous-grilles sur lesquelles on va appliquer l'algorithme séquentiel. Puis on essaie de prouver que le programme parallèle implique le programme séquentiel. Bien évidemment, toutes les obligations de preuves fournies par le prouveur concernant les frontières ne pourront pas être prouvées. Mais ces obligations de preuves définissent la post-condition de la colle, si la spécification est bien conçue. Malheureusement, il est difficile, dans la plupart des cas, de conclure qu'une spécification est correcte sans en avoir fait la preuve. Or nous ne souhaitons pas faire la preuve que notre spécification est correcte puisque cela revient à prouver notre programme séquentiel et ce n'est pas notre objectif. Donc il faut se contenter de l'intuition.
Voici un exemple sur notre simulation : supposons que nous n'ayons pas d'idée quant à la post-condition nécessaire pour assembler les différents résultats obtenus en parallèle. Si on essaie de prouver le théorème `EQUI`, sans les deux lignes avec `VerticalGlueing` et `HorizontalGlueing`, avec PVS, on obtient plusieurs obligations de preuves non prouvées. Par exemple, l'obligation suivante (c'est un séquent) dans laquelle nous avons caché les formules non pertinentes.

```
[-1]    x!1 = floor(x!1 / band_w!1) * band_w!1 - 1 + band_w!1
[-2]    y!1 >= 0
[-3]    y!1 < height!1 - 1
[-4]    HLink!1(x!1, y!1)
  |-------
[1]    Spin2p!1(x!1, y!1) = Spin2p!1(x!1 + 1, y!1)
```

Après avoir supprimé les formules inutiles pour la lisibilité du séquent, on peut l'interpréter de la manière suivante : Pour tout `x` et `y` (le caractère `!` suivi d'un numéro représente une variable skolémisée), si `x` est sur une frontière[18] et si `HLink(x,y)` est vrai[19], alors `Spin2p(x,y)=Spin2p(1+x,y)`. Ainsi on retrouve la définition de `VerticalGlueing`.

9.4 Preuve du second système de transition de phase

Nous allons maintenant prouver que la parallélisation présentée dans le chapitre 7 est correcte. Dans un premier temps, nous allons réaliser une abstraction de notre code séquentiel (page 68). L'abstraction que nous avons appliquée ne conserve pas l'ordre d'exécution des boucles puisque nous modélisons les différentes boucles par un `FORALL`. Dans la première boucle, nous remplacons la racine carrée par une constante. Dans la seconde boucle, nous choisissons de supprimer la boucle la plus imbriquée et le calcul de la variable `prod`. Ainsi l'abstraction rend compte des modifications des tableaux `v1` et `v2`. Nous utilisons toujours la lettre « p » pour indiquer qu'une

18. C'est-à-dire que `x = floor(x / band_w) * band_w - 1 + band_w`.
19. C'est-à-dire qu'il y a un lien.

variable est primé au sens de Lamport, c'est-à-dire qu'elle représente l'état après exécution de la formule.

```
v1,v1p,v2,v2p : var [nat -> nat]
cte : TYPE =x:nat|x>2 CONTAINING 3
kte,kml,kmr : cte
Nq : nat = kml*kmr
kmax:nat = Nq*Nq*Nq*Nq
ict,ict2,iA,iB,iB2,iC,itot,proc,i,iv1,iv2: var nat

seq(v1,v1p,v2,v2p) : bool =
 (FORALL iA,iB,ict,ict2,itot:
  iA>=1 and iA<= kml and
  iB>=1 and iB<=kmr and
  ict>=0 and ict<Nq and
  ict2=(ict*Nq+1) and
  itot=iA+(ict2+(iB-1)*Nq*Nq)*kml
  => v1p(itot)=v1(itot)*kte
 )
 and
 (FORALL iA,iB,iC,iv1,iv2:
   iA>=1 and iA<=kml*Nq and
   iB>=1 and iB<=kmr*Nq and
   iC>=1 and iC<=kml*Nq and
   iv2=iC+(iB-1)*kml*Nq and
   iv1=iA+Nq*kml*(iB-1)
   =>
   v2p(iv2)=v2(iv2)+kte*v1p(iv1)
 )
```

Ensuite, nous définissons l'abstraction du programme parallèle. Nous avons choisi de réaliser l'abstraction de la première version du programme parallèle que nous avons obtenu, en raison des optimisations que nous avons apportées ultérieurement. La différence entre l'abstraction du programme séquentiel et du programme séquentiel se situe au niveau des bornes de la variable `iB` qui sont réduites dans la version parallèle, puisque chaque processeur exécute une partie différente de l'ensemble des itérations du programme séquentiel. La variable `proc` représente le numéro du processeur.

```
para(v1,v1p,v2,v2p,proc) : bool =
 (FORALL iA,iB,ict,ict2,itot:
  iA>=1 and iA<=kml and
  iB=proc and
  ict>=0 and ict<Nq and
  ict2=(ict*Nq+1) and
  itot=iA+(ict2+(iB-1)*Nq*Nq)*kml
  => v1p(itot)=v1(itot)*kte
 )
 and (FORALL iA,iB,iC,iv1,iv2:
   iA>=1 and iA<=kml*Nq and
   iB>=(proc-1)*Nq and iB<=proc*Nq and
   iC>=1 and iC<=kml*Nq and
   iv2=iC+(iB-1)*kml*Nq and
   iv1=iA+Nq*kml*(iB-1)
   =>
```

```
v2p(iv2)=v2(iv2)+kte*v1p(iv1)
)
```

Comme chaque processeur travaille sur une partie des tableaux `v1` et `v2` qui lui est propre, il n'est pas nécessaire de définir de colle. Ainsi la preuve que notre parallélisation est correcte se résume à prouver que les différentes post-conditions des parties parallèles impliquent la post-condition du programme séquentiel. La variable `i` représente le nombre de processeur.

```
parallel : THEOREM
  (FORALL i: i>=1 and i<=kmr => para(v1,v1p,v2,v2p,i)) => seq(v1,v1p,v2,v2p)
```

L'appel de la fonction `mpi_allgather` permet à chaque processeur de diffuser aux autres processeurs la partie du tableau `v2` qu'il a calculé. Ainsi, tous les processeurs possèdent une copie de ce tableau. L'abstraction que nous avons choisie englobe cet appel de fonction MPI puisque nous ne séparons pas le tableau `v2`. Cette abstraction aurait donc été la même si nous avions choisi de paralléliser cette application avec de la mémoire partagée.

9.5 Conclusion

Nous proposons une méthode permettant de prouver qu'un programme parallélisé en utilisant la décomposition de domaines est correct en supposant que le programme séquentiel est correct. Le principe repose sur les post-conditions du programme. Il faut prouver que les post-conditions du programme parallèle plus la post-condition de la colle, impliquent la post-condition du programme séquentiel (condition 9.1). La difficulté d'établir la post-condition d'un programme n'est pas écartée, elle constitue toujours un obstacle. La validité de cette post-condition peut être vérifiée, en insérant cette post-condition spécifiée en logique directement dans le code, lors de l'exécution.
De plus, nous pouvons paralléliser une application à l'aide d'un prouveur de théorèmes et celui-ci, grâce aux obligations de preuves, peut nous aider à définir la post-condition de la colle si nous ne l'avons pas spécifiée auparavant.
Nous avons vérifié que les deux parallélisations utilisant une décomposition de domaines que nous avons effectuées, c'est-à-dire les deux simulations de transitions de phase, sont correctes.

10

Compilation de programmes UNITY en Fortran parallèle

10.1 Introduction

La conception de programmes parallèles a été largement influencée par le cadre UNITY introduit par Chandy et Misra [Chandy et Misra, 1988]. Le succès d'UNITY est principalement dû à la simplicité de notation de programme et à l'intégration d'une logique de programmation (un fragment de logique temporelle linéaire). Le travail de ce chapitre concerne la traduction d'une variante d'UNITY, appelée UNITY parallèle [Van de Velde, 1994], en un Fortran parallèle, exécuté sur une machine à mémoire partagée SGI utilisant OpenMP. Comme nous produisons du code pour une machine parallèle, l'Origin 2000, nous devons étudier l'efficacité du code produit tout en maintenant la correction du programme initial UNITY.
Van Velde [Van de Velde, 1994] a introduit la notation de programmes parallèles UNITY et l'a utilisée sur de nombreux exemples. Notre travail a d'abord consisté à définir le langage de programmation et à construire un analyseur syntaxique. Nous avons effectué des traductions à la main, étudié certains exemples de Van Velde tout en gardant en tête l'efficacité du code obtenu. Finalement, nous avons construit un compilateur d'UNITY parallèle en Fortran avec OpenMP, écrit en Java, et une interface Java permettant de tester le code résultant de la compilation.
Ce travail a été entrepris pour montrer qu'il est possible d'obtenir un code performant à partir d'une spécification formelle. L'objectif à long terme est de pouvoir disposer d'un environnement dans lequel on effectue des preuves de spécifications que l'on peut ensuite traduire en code exécutable pour des machines parallèles.

10.2 Quelle version d'UNITY ?

UNITY [Chandy et Misra, 1988] a été développé pour fournir un cadre dans lequel on peut définir des programmes, des propriétés et des raffinements, tout en gardant une notation uniforme.
Dans son livre [Van de Velde, 1994], Eric Van de Velde explique que l'utilisation de bonnes notations aide à réduire les problèmes dans le développement de programmes parallèles. Mais, bien sûr, cela n'évite pas tous les problèmes. Il choisit d'écrire ses programmes avec une version modifiée d'UNITY dans laquelle l'indéterminisme n'est pas autorisé. Ainsi, l'implantation sur une machine parallèle se révèle plus efficace.
Pour des raisons de simplicité, il ajoute l'instruction de réduction suivante :

$t :=< +m : 0 \leq m \leq M :: x[m] >$

qui réalise la somme de $x[m]$. L'opérateur + peut être remplacé par l'opérateur « · » et nous pouvons calculer un produit avec une seule instruction. En essayant d'analyser automatiquement les programmes écrits par Van de Velde, nous avons rencontré des problèmes avec la grammaire, c'est pourquoi nous avons choisi de la modifier légèrement. Nous résumons ici les différences obtenues avec la grammaire de Van de Velde.

- Nous ne permettons pas les affectations multiples car elles sont inefficaces avec des variables scalaires lorsqu'il y a peu d'affectations (elles pourraient être intéressantes avec des opérations matricielles, mais dans ce cas, la parallélisation de ces opérations est plus efficace et puissante).
- Toutes les opérations écrites avec des notations mathématiques sont représentées sous leur forme informatique. Donc, $\sqrt{a}$ s'écrit $sqrt(a)$, x^T s'écrit $transp(x)$ (c'est la transposition d'un vecteur). Toutes les multiplications sont explicites, donc Ax s'écrit $A{\cdot}x$ et x^2 s'écrit $x{\cdot}x$, pour simplifier. La construction d'un complexe $a+ib$ s'écrit $cmplx(a,b)$. $\overline{a}$ s'écrit $conjg(a)$. La partie imaginaire de a s'écrit $imag(a)$ et la partie réelle de a est $real(a)$.

10.3 Implantation

Pour exécuter des programmes UNITY, nous produisons du code Fortran 77 avec des directives de compilations OpenMP, et ensuite nous le compilons sur l'Origin 2000. La traduction du code UNITY vers le Fortran est réalisée à l'aide du générateur de compilateur Java [Sun Microsystems, 1998]. Cet outil est très simple à utiliser et il définit une grammaire (une LL(1) par défaut mais elle peut être LL(k) à certains endroits) en construisant des relations entre chaque terme de la grammaire. Ensuite le générateur de compilateur construit un squelette de chaque classe Java. On doit spécifier pour chaque classe comment réaliser la traduction. Le travail le plus difficile consiste à générer un code afin de manipuler correctement les opérations matricielles. Nous allons voir le travail réalisé pour chaque section d'un programme UNITY.
La section *declaration* est une simple traduction des types UNITY en types Fortran. Nous donnons un exemple dans la figure 10.1.

```
lamb : complex;                    Complex lamb
v : array [100] of complex;        Complex v(0:100)
A :  array [100 X 100] of real;    Real A(0:100,0:100)
tau : real                         Real tau
```

FIG. 10.1 – *Traduction d'une déclaration UNITY en une déclaration Fortran*

La section *initially* est similaire à la section *assign*. Si nous considérons une initialisation scalaire, vectorielle ou matricielle, le code produit est le même que pour la section *assign*. En fait, on distingue ces deux sections seulement pour les séparer ; c'est pourquoi la syntaxe se trouve légèrement différente mais la sémantique reste la même. Dans certains cas, la section *initially* est supprimée parce que nous avons besoin d'affecter des valeurs aléatoires, et ce type d'opérations doit être réalisé dans la section *assign*.
Dans la section *assign*, nous avons à traiter 4 types d'instruction : les affectations, les boucles, les conditions et les réductions.
Dans la partie affectation, il faut différencier les affectations scalaires, des affectations vectorielles, et des affectations matricielles : dans les deux derniers cas, nous devons générer automatiquement le code et capturer les éventuelles erreurs sur ces opérations. Donc une affectation scalaire est

traduite simplement et nous ajoutons les sous-routines Fortran qui traitent tous les cas des opérations matricielles et vectorielles (multiplication de matrice-matrice, matrice-vecteur, produit scalaire de vecteur). Nous donnons un exemple d'une sous-routine de produit matrice-vecteur avec la figure 10.2.

```
      Subroutine MVMult
     *  (M,V,size_i,size_j,VRes)
      Integer size_i,
     *  size_j,i,j,k
      Real M(0:size_i,0:size_j),
     *  V(0:size_j),VRes(0:size_i),
     *  Sum

!$OMP PARALLEL DO PRIVATE(i,j,k,Sum)
      Do i=0,size_i
        Do j=0,size_j
          Sum=0
          Do k=0,size_j
            Sum=Sum+M(i,k)*V(k)
          Enddo
          VRes(i)=Sum
        Enddo
      Enddo
      End
```

FIG. 10.2 – *produit matrice-vecteur*

Les boucles (représentées par des quantificateurs universels) sont traduites facilement et nous avons seulement à spécifier le statut de chaque variable pour OpenMP ; cette tâche est faite automatiquement. Comme la sémantique de l'opérateur || d'UNITY impose que les instructions qui sont imbriquées dans la boucle peuvent être exécutées en parallèle, nous supposons que la boucle est parallélisable et que le développeur de la spécification s'en est assuré. Nous traitons ce cas en générant une directive de compilation qui permet à la boucle d'être exécutée en parallèle. Si par contre, on rencontre l'opérateur « ; » dans la déclaration d'une boucle, on génère le code sans directive. Nous donnons un exemple d'une boucle parallèle :

```
<|| j : i+1 <= j < n ::
       ar[i,j] := ar[i+1,j]>
```

et sa traduction en Fortran :

```
!$OMP PARALLEL DO PRIVATE(j)
      Do j = i+1 , n-1
        ar(i,j) = ar(i+1,j)
      Enddo
```

Afin de simplifier l'utilisation de notre compilateur, nous avons construit une interface Java qui permet d'ouvrir un fichier, de l'éditer et de le sauvegarder. Les mots clés d'UNITY sont affichés avec une couleur différente pour une meilleur lecture des programmes. Si l'utilisateur considère que le fichier UNITY est convenable, il peut le traduire en Fortran. Dans ce cas, s'il n'y a pas d'erreurs syntaxiques dans la spécification, l'utilisateur peut soumettre son travail à

la machine parallèle. Une boîte de dialogue permet de choisir l'option que l'on souhaite utiliser pour le compilateur Fortran, quel navigateur utiliser pour lire l'aide, et le nombre de processeurs souhaité pour chaque exécution. Après l'exécution du programme, le résultat de celui-ci est affiché dans une fenêtre. La figure 10.3 montre une capture écran de l'interface.

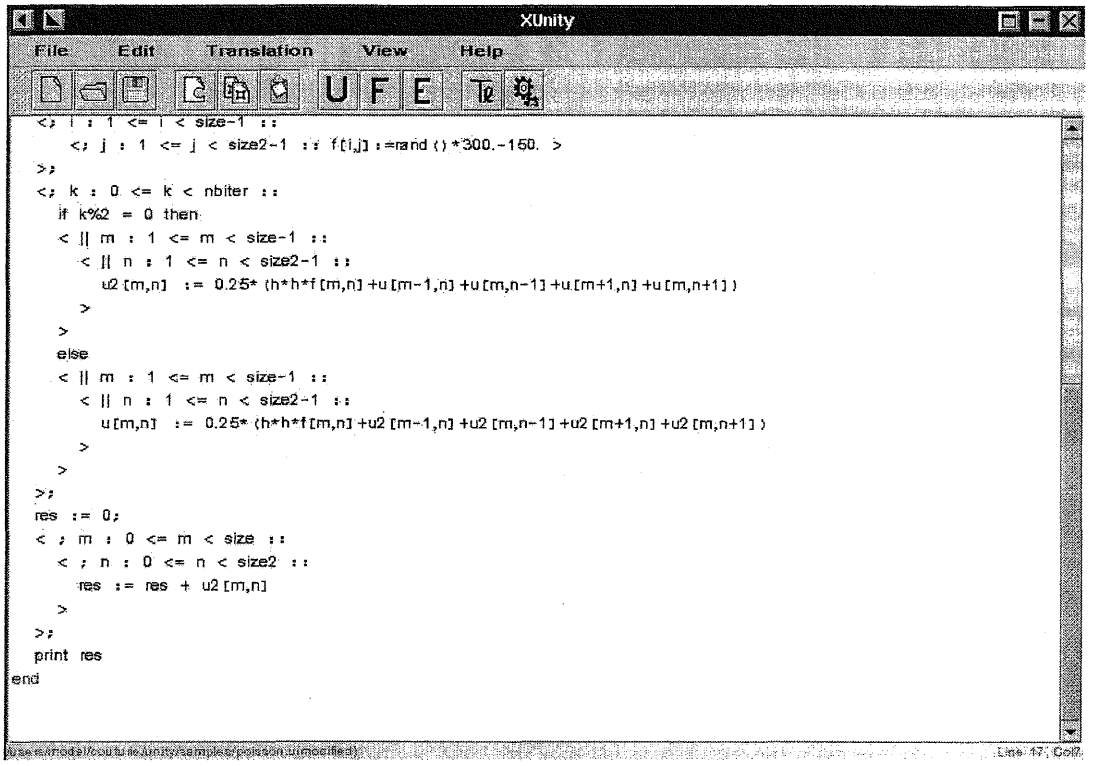


FIG. 10.3 – *Interface du compilateur Fortran UNITY*

10.4 Preuve informelle que notre implantation est correcte

Pour prouver que notre implantation est correcte, nous allons procéder par induction sur la structure d'un programme UNITY que notre compilateur utilise. La section *declaration* est simplement une traduction des types d'UNITY vers les types de Fortran. Comme, à tout type UNITY utilisé, il existe un type Fortran équivalent, on est sûr que cette traduction est correcte. La section *initially* est identique à la section *assign* c'est pourquoi nous nous occupons directement de la section *assign*. Nous avons vu qu'une telle section comporte 4 types d'instructions. Les affectations sont différenciées en affectations scalaires, vectorielles et matricielles. Pour les opérations simples, la traduction ne pose pas de difficulté. Pour les opérations vectorielles et matricielles, nous avons défini des sous-routines Fortran adéquates pour les traiter en parallèle. Les boucles sont le deuxième type d'instruction que l'on peut rencontrer. Lorsqu'une boucle est séquentielle la traduction est évidente. Si l'opérateur || est utilisé, comme nous supposons que son utilisation a été prouvé, nous insérons une directive parallèle qui déclare en `PRIVATE` toutes les variables scalaires et en `SHARED` toutes les autres variables. Si on veut s'assurer que l'opérateur || est utilisé correctement, c'est-à-dire que la boucle est parallélisable, on peut utiliser le raffinement d'UNITY pour prouver qu'un programme parallèle est correcte. Les conditions sont traduites simplement

d'UNITY vers le Fortran. Finalement les réductions sont traitées avec l'opérateur `REDUCTION` défini avec OpenMP qui possède la même sémantique que celle proposée par Van de Velde.

10.5 Résultats

Toutes les expérimentations que nous avons menées, ont été réalisées sur l'Origin 2000 alors que d'autres personnes l'utilisaient. La conséquence est que, suivant la charge de la machine, on constate quelques variations dans le temps d'exécution. Tous les temps sont exprimés en secondes.

- Le problème de N-Corps avec 5000 corps et 100 itérations (tous les corps interagissent entre eux).

nombre de processeurs	temps	accélération
1	980	1
4	252	3.88
16	69	14.20

- Résolution par la méthode de Gauss Seidel d'une matrice 4000*4000.

nombre de processeurs	temps	accélération
1	6644	1
4	1857	2.50
16	741	8.96

- Solveur de Poisson avec une matrice 6000*3000 et 400 itérations.

nombre de processeurs	temps	accélération
1	7217	1
4	2020	3.57
16	647	11.15

À la vue de ces exemples, nous obtenons globalement des résultats satisfaisants (il faut garder en tête que le code est généré automatiquement et qu'on peut encore l'optimiser à la main si on le souhaite). Nous donnons le code de la résolution de Gauss Seidel afin d'illuster la structure d'un code UNITY.

```
program GaussSeidel
declare
  i, j, k : integer;
  n = 4000;
  ar : array[n+1 X n] of real
assign
  {initialization of the matrix}
  randinit(234);
  <; i : 0 <= i < n+1 ::
    <; j : 0 <= j < n ::
         ar[i,j]:=rand()*10000+1
    >
  >;
  {triangularization}
  <; i : 0 <= i < n-1 ::
    <|| j : i+1 <= j < n+1 ::
      <; k : i+1 <= k < n ::
         ar[j,k] := (ar[j,k] -
           (ar[i,k]*ar[j,i]) / ar[i,i])
```

```
      >
    >;
    <|| j : i+1 <= j < n :: ar[i,j] := 0 >
  >;
  {resolution}
  <; i : n > i >= 0 ::
    ar[i,i] := ar[i+1,i]/ar[i,i];
    <|| j : 0 <= j < i ::
      ar[i,j] := ar[i+1,j]-ar[i,j]*ar[i,i]
    >;
    <|| j : i+1 <= j < n ::
      ar[i,j] := ar[i+1,j]
    >
  >;
  <; i : 0 <= i < n ::
    print ar[0,i]
  >
end
```

Dans l'annexe B, nous donnons le code Fortran produit par la traduction de la spécification de la résolution de Gauss-Seidel, à titre d'exemple.

10.6 Conclusion

La notation d'UNITY parallèle est plus abstraite que le code Fortran obtenu et le résultat, à travers un code plus lisible, peut être dérivé en utilisant la méthodologie d'UNITY. Elle a l'avantage de lier la notation assez proche de la logique à la preuve. Au niveau performance, si le travail qui doit être effectué dans chaque boucle est suffisamment important, les résultats sont bons. Ce travail montre toutefois qu'on peut partir d'une spécification formelle et obtenir un programme parallèle relativement efficace. Il est clair que les concepts de haut niveau d'OpenMP y sont pour quelque chose. C'est pourquoi nous pensons qu'il est possible de rapprocher le parallélisme et les méthodes formelles. Ce travail illustre une possibilité de rapprochement.

Lors de ce travail nous avons cherché à rapprocher un outil de développement de preuves avec cette version d'UNITY. L'intérêt de cette version d'UNITY est d'être plus proche d'un langage d'implantation que les autres langages de spécification. L'inconvénient réside dans la difficulté que l'on a pour lier cette version d'UNITY à PVS dans notre cas. Les principales difficultés sont les suivantes. Cette version d'UNITY utilise des instructions qui sont exécutées séquentiellement ou de manière parallèle lorsqu'on utilise l'opérateur parallèle. Or, PVS et de nombreux langages de spécification ont plutôt tendance à prendre des fonctions récursives pour modéliser une boucle. L'enchaînement entre toutes les itérations est donc complexes à gérer. Si on prend le problème dans l'autre sens. On pourrait se dire qu'il faut utiliser des fonctions récursives dans UNITY. Mais dans ce cas, c'est le Fortran qui ne va pas pouvoir les exécuter dans la version 77 et les langages itératifs utilisés pour programmer des applications parallèles (Fortran, C, C++) n'utilisent en général que peu de fonctions récursives.

Conclusions et perspectives

Le travail réalisé dans cette thèse a pour motivation de montrer que les méthodes formelles sont utilisables pour développer des programmes parallèles, mais également de cerner les limites actuelles qu'elles possèdent.
En ce qui concerne le parallélisme, nous avons parallélisé trois simulations utilisées par des physiciens et des chimistes. Par le biais de ces parallélisations, nous avons acquis une certaine expérience qui nous permet d'affirmer que le dialogue avec les personnes collaboratrices est très important. En ce qui concerne les bibliothèques de communications, nous avons utilisé suivant les besoins, OpenMP, MPI ou les deux à la fois. Cette dernière combinaison semble très intéressante à l'avenir. En effet, le développement de réseaux de machines multiprocesseurs à mémoire partagée semble une piste intéressante à tous niveaux (prix, performance, développement d'applications) pour le futur. Donc il est possible d'utiliser MPI pour communiquer entre les machines multiprocesseurs et OpenMP pour utiliser la mémoire partagée de chaque machine.

Suivant la nature d'une application, l'architecture matérielle et l'environnement de développement choisis, nous sommes amenés à utiliser des schémas ou archétypes différents pour la parallélisation d'un programme.
Si nous choisissons d'utiliser la décomposition de domaines, nous avons défini une méthode qui permet de prouver qu'une parallélisation est correcte, ou qui permet de trouver les conditions permettant d'obtenir une bonne parallélisation, à l'aide d'un prouveur de théorèmes (dans notre cas PVS). La difficulté consiste à déterminer une abstraction du programme séquentiel et du programme parallèle qui permette de prouver la correction de la parallélisation.
Concernant la parallélisation de boucles avec OpenMP, nous n'avons pas réussi à modéliser comment prouver que les transformations apportées au programme séquentiel sont correctes. La finesse des transformations induit des preuves beaucoup trop compliquées au regard de la simplicité d'utilisation d'OpenMP.

Le développement d'un algorithme peut très bien être effectué à l'aide d'un prouveur. Nous commençons par prouver des propriétés et finalement nous obtenons une spécification du programme. Nous avons illustré cette démarche incrémentale pour construire une spécification du tri bitonique, tri facilement parallélisable. Finalement, nous pouvons traduire à la main l'algorithme obtenu dans le langage C++ en utilisant les directives de compilation pour la parallélisation.
Comme cette traduction à la main nous semble inadaptée, nous avons construit un compilateur pour effectuer le passage d'une spécification d'un programme parallèle, au code parallèle. Ainsi nous obtenons directement du code Fortran avec OpenMP à partir d'une spécification UNITY.

Après les travaux que nous avons menés, nous pouvons nous poser la question suivante : pourquoi les développeurs n'utilisent pas davantage les méthodes formelles dans leurs développements, surtout pour les programme parallèles? Il est clair que nous avons tout à y gagner après avoir

accompli un effort d'apprentissage qui est tout de même long, il ne faut pas le négliger, car les méthodes formelles nécessitent des connaissances souvent pointues en comparaison des connaissances requises en vue de développer un programme parallèle. De nombreux travaux ont pour but d'automatiser et de simplifier l'utilisation des méthodes formelles. Les outils actuels, performants, ne sont pas à la portée de tout le monde. Il faudrait peut-être étudier davantage l'utilisation des méthodes formelles et des outils qui en dérivent dans la formation universitaire, afin d'en donner les bases à un plus grand nombre de personnes.

Néanmoins, les collaborations que nous avons menées avec les physiciens et les chimistes, nous ont permis de prendre conscience du fossé qui sépare le monde des informaticiens de celui des non-informaticiens envers le parallélisme. Les simulations numériques des non-informaticiens sont très longues. La simulation de dynamique moléculaire que nous avons parallélisée a été exécuté plus d'une année en séquentiel. Nous comprenons donc pourquoi la possibilité de réduire le temps d'une simulation est primordiale. L'autre intérêt du parallélisme est d'offrir la possibilité de calculer une simulation plus importante car une machine parallèle possède, en général, beaucoup plus de mémoire qu'une machine séquentielle, même si le fait de paralléliser une application a pour effet de dupliquer tout de même certaines données.
La parallélisation d'une application est très différente suivant que nous connaissons l'application ou non. Si nous ne connaissons pas une application avant de tenter de la paralléliser, le langage de programmation utilisé, la longueur du code ainsi que la structure de celui-ci sont des points importants à surveiller. Nous pensons qu'il est intéressant de proposer la méthode de parallélisation choisie aux personnes qui ont développé le programme séquentiel, afin qu'il puisse la comprendre et la critiquer. Une étape de tests pour analyser la méthode de parallélisation sur un modèle simplifié, permet de valider la démarche retenue. L'autre avantage des discussions avec les collaborateurs est de leur apprendre, ou au moins de leur donner des notions des environnements parallèles. Ainsi, ultérieurement ils pourront simplifier une éventuelle parallélisation en structurant correctement leurs programmes.

En perspective, nos travaux peuvent être prolongés de diverses manières. Pour développer un programme parallèle correct, il faudrait avoir la possibilité d'effectuer des preuves de propriétés directement dans le langage d'UNITY. PVS nous semble adapté pour ce genre de travail, mais il faut néanmoins définir une stratégie qui permettrait de construire la spécification, dans le langage de PVS, d'un programme UNITY. Notons que Paulson [Paulson, 1999] vient juste de définir en Isabelle une technique pour mécaniser les preuves de UNITY. Ensuite, nous pourrions compiler la spécification grâce à notre compilateur en Fortran pour une machine parallèle utilisant OpenMP. Ainsi, nous aurions un moyen d'obtenir des programmes parallèles efficaces et corrects.
L'utilisation d'UNITY semble intéressante mais d'autres langages de spécification sont peut-être mieux adaptés dans certains cas. Un langage de spécification utilisé dans un prouveur restreint à un certain type de fonctionnalités qui permettent facilement la transformation vers un exécutable parallèle pourrait peut-être se révéler être la solution.
En tout cas, le fait de générer du code en utilisant OpenMP nous semble de loin la meilleure solution au niveau performance, mise en œouvre, et en raison de l'approche de haut niveau qu'il procure.
De même les archétypes utilisés par Massingill sont très intéressants car ils sont proches du concept de programmation avec composants. Actuellement de nombreux outils sont développés pour réutiliser des composants pour la programmation séquentielle. La difficulté pour développer des composants pour des programmes parallèles réside dans la diversité des machines, des

langages, des moyens de communications et se trouve être la conséquence du peu de personnes compétentes en parallélisme en regard du nombre de personnes qui développent des programmes séquentiels. Ces facteurs sont la cause actuelle de la limitation des travaux qui ont pour objectif d'établir un recueil d'archétypes. Un tel recueil est un moyen de diminuer le temps de développement d'un programme parallèle ainsi qu'un moyen d'éliminer les erreurs de programmation dès lors que l'archétype est correct. Mais pour qu'il est une chance d'exister il faut que de nombreuses personnes en aient la volonté. Or à ce jour, ce projet n'a pas l'air d'avoir une grande notoriété.

A

Première annexe

A.1 Comparaison d'un programme parallélisé avec OpenMP et MPI

Pour donner une idée des modifications introduites afin de paralléliser avec les deux approches citées ci-dessus, voici deux codes qui réalisent une multiplication de matrice par un vecteur.
Les deux programmes n'ont pas été écrits à partir du même programme séquentiel et n'ont pas été écrits par les mêmes personnes. Nous avons parallélisé le programme avec OpenMP et nous avons repris le programme utilisant MPI dans un fichier d'exemples de parallélisation avec MPI, disponible à l'adresse suivante :
http ://www.ncsa.uiuc.edu/SCD/Hardware/CommonDoc/MessPass/MPImatmult.html.
La différence de taille des deux codes est flagrante, d'autant plus que la différence de speed-up existe aussi. Nous ne précisons pas de temps de mesure, mais il est évident qu'en travaillant avec la mémoire partagée pour cet exemple, nous gagnons beaucoup de temps en utilisant OpenMP.

A.1.1 Parallélisation en utilisant OpenMP

```
Program MulMat
  Parameter   (Nb = 2000)
  Real*8      Mata(1:Nb,1:Nb),Matb(1:Nb,1:Nb)
  Real*8      Matc(1:Nb,1:Nb),res
  Integer     a,b,c
  Integer     e,f
  Real        t1,t2,ttime(2),d


!$OMP PARALLEL DO PRIVATE(a,b)
  Do a=1,Nb
    Do b=1,Nb
      Mata(a,b)=a*2.3444/b
      Matb(a,b)=1.002*a+b/7
      Matc(a,b)=0
    Enddo
  Enddo
```

```
  t1=Etime(ttime)


!$OMP PARALLEL DO PRIVATE(a,b,c)
  Do a=1,Nb
    Do b=1,Nb
      Do c=1,Nb
        Matc(a,b)=Mata(a,c)*Matb(c,b)
      Enddo
    Enddo
  Enddo

  t2=Etime(ttime)

  res=0
!$OMP PARALLEL DO PRIVATE (a,b) SHARED(Matc) REDUCTION(+:res)
  Do a=1,Nb
    Do b=1,Nb
      res=res+Matc(a,b)
    Enddo
  Enddo

  Write (*,*) 'fin',t2-t1,'sec',' res',res
  end
```

A.1.2 parallélisation en utilisant MPI

```
ccccccccccccccccccccccccccccccccccccccccccccccccc
c      Matrix Multiplication MPI Program       c
c      For this simple version, # of procssors c
c      equals # of columns in matrix           c
c                                              c
c      To run, mpirun -np 4 a.out              c
ccccccccccccccccccccccccccccccccccccccccccccccccc

      include 'mpif.h'

      parameter (ncols=4, nrows=4)
      integer a(ncols,nrows), b(ncols,nrows), c(ncols,nrows)
      integer buf(ncols),ans(nrows)
      integer myid, root, numprocs, ierr, status(MPI_STATUS_SIZE)
      integer sender, count

      call MPI_INIT(ierr)
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
```

```
      if(numprocs.ne.4) then
        print *, "Please run this exercise on 4 processors"
        call MPI_FINALIZE(ierr)
        stop
      endif

      root = 0
      tag = 100
      count = nrows*ncols

c     Master initializes and then dispatches to others
      IF ( myid .eq. root ) then

         do j=1,ncols
            do i=1,nrows
               a(i,j) = 1
               b(i,j) = j
            enddo
         enddo

c        send a to all other process
         call MPI_BCAST(a,count,MPI_INTEGER,root,MPI_COMM_WORLD,ierr)

c        send one column of b to each other process
         do j=1,numprocs-1
            do i = 1,nrows
               buf(i) = b(i,j+1)
            enddo
         call MPI_SEND(buf,nrows,MPI_INTEGER,j,tag,MPI_COMM_WORLD,ierr)
         enddo

c        Master does his own part here
         do i=1,nrows
            ans(i) = 0
            do j=1,ncols
               ans(i) = ans(i) + a(i,j) * b(i,1)
            enddo
         c(i,1) = ans(i)
         enddo

c        then receives answers from others

         do j=1,numprocs-1
            call MPI_RECV(ans, nrows, MPI_INTEGER, MPI_ANY_SOURCE,
     $           MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)

            sender = status(MPI_SOURCE)
            do i=1,nrows
```

```
                c(i,sender+1) = ans(i)
             enddo

          enddo

          do i=1,nrows
             write(6,*)(c(i,j),j=1,ncols)
          enddo

       ELSE

c         slaves receive a, and one column of b, then compute dot product
          call MPI_BCAST(a,count,MPI_INTEGER,root,MPI_COMM_WORLD,ierr)

          call MPI_RECV(buf, nrows, MPI_INTEGER, root,
     $         MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)

          do i=1,nrows
             ans(i) = 0
             do j=1,ncols
                ans(i) = ans(i) + a(i,j) * buf(j)
             enddo
          enddo

          call MPI_SEND(ans,nrows,MPI_INTEGER,root,0,MPI_COMM_WORLD,ierr)

       ENDIF

       call MPI_FINALIZE(ierr)

       stop
       end
```

B

Deuxième annexe

Voici la traduction en Fortran de la spécification UNITY du chapitre 10.

```
Program GaussSeidel
Implicit none
external rand
real*8 rand


c DECLARE
Integer i, j, k
parameter n = 4000
Real ar(0:4000,0:3999)

c ASSIGN
call srand(234)
Do i = 0 , n+1-1
  Do j = 0 , n-1
    ar(i,j) = rand()*10000+1
  Enddo
Enddo
Do i = 0 , n-1-1
!$OMP PARALLEL DO PRIVATE(j, k)
  Do j = i+1 , n+1-1
    Do k = i+1 , n-1
      ar(j,k) = (ar(j,k)-(ar(i,k)*ar(j,i))/ar(i,i))
    Enddo
  Enddo
!$OMP PARALLEL DO PRIVATE(j)
  Do j = i+1 , n-1
    ar(i,j) = 0
  Enddo
Enddo
Do i = n-1 , 0 , -1
  ar(i,i) = ar(i+1,i)/ar(i,i)
!$OMP PARALLEL DO PRIVATE(j)
```

```
  Do j = 0 , i-1
    ar(i,j) = ar(i+1,j)-ar(i,j)*ar(i,i)
  Enddo
!$OMP PARALLEL DO PRIVATE(j)
  Do j = i+1 , n-1
    ar(i,j) = ar(i+1,j)
  Enddo
Enddo
Do i = 0 , n-1
  Print *,'ar(0,i) = ', ar(0,i)
Enddo
Print *,'Program GaussSeidel OK'
End
```

C

Troisième annexe

Dans cette annexe, nous donnons la spécification entière des propriétés du tri bitonique.

```
bitonic_sort  : THEORY

  BEGIN

  Arra : TYPE = [nat->nat]
  n,x,y,nb,nbp,i,j,low,hi,pos,s,sam,sp,spam,nx,ny:VAR nat
  A,SA,Ap,SAp : VAR Arra

  BitonicMin(A,Ap,low,nbp) : bool =
    FORALL x: x>=low and x<low+nbp => Ap(x) = min(A(x),A(x+nbp))

  BitonicMax(A,Ap,low,nbp) : bool =
    FORALL x:  x>=low and x<low+nbp =>
      Ap(x+nbp) = max(A(x),A(x+nbp))


  IncreasingList(A,low,hi) : bool = FORALL x,y:x>=low and y<=hi and
x<=y => A(x)<=A(y)

  DecreasingList(A,low,hi) : bool = FORALL x,y:x>=low and y<=hi and
x<=y => A(x)>=A(y)

  BitonicList(A,low,hi,i) : bool = i>low and i<=hi and
IncreasingList(A,low,i-1) and DecreasingList(A,i,hi)

  IncreasingList2(A,low,hi,nbp,s) : bool = FORALL x,y:x>=low and y<=hi and
x<=y => FORALL nx,ny:
nx=IF x+s>=low+nbp THEN x+s ELSE x+s+nbp ENDIF and
ny=IF y+s>=low+nbp THEN y+s ELSE y+s+nbp ENDIF =>
A(nx)<=A(ny)
```

```
  DecreasingList2(A,low,i,hi,nbp,s) : bool = FORALL x,y:x>=i and y<=hi and
x<=y => FORALL nx,ny:
nx=IF x+s>=low+nbp THEN x+s ELSE x+s+nbp ENDIF and
ny=IF y+s>=low+nbp THEN y+s ELSE y+s+nbp ENDIF =>
A(nx)>=A(ny)

  BitonicList2(A,low,hi,i,nbp,s) : bool = s>=0 and s<nbp and (( i>low
and i<=hi and IncreasingList2(A,low,i-1,nbp,s) and
DecreasingList2(A,low,i,hi,nbp,s)) or
DecreasingList2(A,low,low,hi,nbp,s) or IncreasingList2(A,low,hi,nbp,s))

   Pivot(A,low,hi,nbp,j) : bool = j>low and j<=low+nbp-1 and
A(j-1)<=A(j-1+nbp) and A(j)>=A(j+nbp) and
(FORALL x:x>=low and x<=j-1 => A(x)<=A(x+nbp)) and
(FORALL x:x>=j and x<=low+nbp-1 => A(x)>=A(x+nbp))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% lemmes pour les places de i et j
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    TRY_0: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)

    TRY_1: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=j and x<=i-1 => A(x)>=A(j)

    TRY_2: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)

    TRY_3: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
  FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)

    TRY_4: LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j)
=>
```

```
(
 (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
 (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
 (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
 (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp))
)

    TRY_5: LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
(
(FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
(FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
(FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
(FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp))
)
=>
 (i>=j or ( A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i))

    TRY_6: LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
(
(FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
(FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
(FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
(FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp))
)
=>
 (i<=j+nbp or (A(i-1)=A(i-nbp-1)  and A(i)<=A(i-1) and i>j+nbp))


Place_ij(A,i,j,nbp): bool =
 (i>=j or (A(i)=A(i+nbp) and A(i)>=A(i-1) and j>i)) and
 (i<=j+nbp or (A(i-1)=A(i-nbp-1) and A(i)<=A(i-1) and i>j+nbp ))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% La première liste obtenue après l'algorithme est une liste bitonique
% simple
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  BIT_0 : LEMMA
hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1 and
(EXISTS i,j:
  BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
  (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and
  (FORALL x:x>=j and x<=i-1 => A(x)>=A(j)) and
  (FORALL x:x>=i and x<=j+nbp-1 => A(x)>=A(j+nbp-1)) and
  (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
```

```
  Place_ij(A,i,j,nbp)
) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,hi,nbp)
=>
  EXISTS j:BitonicList(Ap,low,low+nbp-1,j)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% La seconde liste obtenue après l'algorithme est une permuation circulaire
% gauche du cas simple, c'est le cas général
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

BIT_1 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and nbp>=1
and (EXISTS i,j:BitonicList(A,low,hi,i) and Pivot(A,low,hi,nbp,j) and
(FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1)) and (FORALL x:x>=j and
x<=i-1 => A(x)>=A(j)) and (FORALL x:x>=i and x<=j+nbp-1 =>
A(x)>=A(j+nbp-1)) and (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp))
and s=j-low and Place_ij(A,i,j,nbp)) and
BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => EXISTS
j:BitonicList2(Ap,low,low+nbp-1,j,nbp,s)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Propriétés des listes qui comportent un décalage
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

R : var PRED[[nat,nat]]

SHIFT(A,SA,hi,low,nb,nbp,sam,s,R) : bool= ((hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 ar
nbp>=1) and s>=0 and s<nb and (FORALL x: x>=low and x<=hi and sam=IF
x+s>hi THEN s-nb ELSE s ENDIF and A(x)=SA(x+sam) )) => (FORALL x:x>=low
and x<low+nbp and R(A(x),A(x+nbp)) => IF x+sam<low+nbp THEN
R(SA(x+sam),SA(x+sam+nbp)) else R(SA(x+sam),SA(x+sam-nbp)) ENDIF)

SHIFT_0 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,>=)

SHIFT_1 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,<=)

SHIFT_2 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,>)

SHIFT_3 : LEMMA SHIFT(A,SA,hi,low,nb,nbp,sam,s,<)

SHIFT_PROP(A,SA,low,nbp,sam) : bool =
 (FORALL x:x>=low and x<low+nbp and A(x)>=A(x+nbp) => IF x+sam<low+nbp
THEN SA(x+sam)>=SA(x+sam+nbp) else SA(x+sam)>=SA(x+sam-nbp) ENDIF)
and
 (FORALL x:x>=low and x<low+nbp and A(x)<=A(x+nbp) => IF x+sam<low+nbp
THEN SA(x+sam)<=SA(x+sam+nbp) else SA(x+sam)<=SA(x+sam-nbp) ENDIF)
```

```
and
  (FORALL x:x>=low and x<low+nbp and A(x)>A(x+nbp) => IF x+sam<low+nbp
THEN SA(x+sam)>SA(x+sam+nbp) else SA(x+sam)>SA(x+sam-nbp) ENDIF)
and
 (FORALL x:x>=low and x<low+nbp and A(x)<A(x+nbp) => IF x+sam<low+nbp
THEN SA(x+sam)<SA(x+sam+nbp) else SA(x+sam)<SA(x+sam-nbp) ENDIF)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Une liste avec décalage est équivalente à une liste sans décalage
% (croissante, décroissante ou bitonique simple)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SHIFT : LEMMA (hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and s>=0 and s<nb and (FORALL x: x>=low and x<=hi and EXISTS
sam: sam=IF x+s>hi THEN s-nb ELSE s ENDIF and A(x)=SA(x+sam) ) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp) and
BitonicMin(SA,SAp,low,nbp) and BitonicMax(SA,SAp,low,nbp)) and (FORALL
sam: SHIFT_PROP(A,SA,low,nbp,sam)) => (FORALL sp: sp=IF s>nbp THEN
s-nbp ELSE s ENDIF => (FORALL x,spam:x>=low and x<low+nbp and spam=IF
x+sp>=low+nbp THEN sp-nbp ELSE sp ENDIF => Ap(x)=SAp(x+spam)))

SHIFT2 : LEMMA (hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and s>=0 and s<nb and (FORALL x: x>=low and x<=hi and EXISTS
sam: sam=IF x+s>hi THEN s-nb ELSE s ENDIF and A(x)=SA(x+sam) ) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp) and
BitonicMin(SA,SAp,low,nbp) and BitonicMax(SA,SAp,low,nbp)) and (FORALL
sam: SHIFT_PROP(A,SA,low,nbp,sam)) => (FORALL sp: sp=IF s>nbp THEN
s-nbp ELSE s ENDIF => (FORALL x,spam:x>=low and x<low+nbp and spam=IF
x+sp>=low+nbp THEN sp-nbp ELSE sp ENDIF => Ap(x+nbp)=SAp(x+nbp+spam)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Une liste croissante après inversion est une liste décroissante
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Swap(A,Ap,low,hi) : bool = (FORALL x:x>=low and x<=hi => Ap(x)=A(hi+low-x))

INC_TO_DEC_LIST :  LEMMA (hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 and  (FORALL x,y: x>=low and y<=hi and x<=y => A(x)<=A(y)) and
Swap(A,Ap,low,hi) ) => (FORALL x,y: x>=low and y<=hi and x<=y => Ap(x)>=Ap(y))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Les éléments de la  première liste obtenue l'algorithme sont inférieurs
% aux éléments de la seconde liste
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

FirstInfSecond(A,low,hi,nbp) : bool =
  FORALL x : x>=low and x<low+nbp => FORALL y : y>=low+nbp and y<=hi
```

```
=> A(x)<=A(y)

FIRST_INF_SECOND : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1
and nbp>=1 and (EXISTS i,j:BitonicList(A,low,hi,i) and
Pivot(A,low,hi,nbp,j) and (FORALL x:x>=low and x<=j-1 => A(x)<=A(j-1))
and (FORALL x:x>=j+nbp and x<=hi => A(x)<=A(j+nbp)) and
 (FORALL x : x>=j and x<low+nbp => A(x)>=A(j-1)) and
(FORALL y : y>=low+nbp and y<=j+nbp => A(y)>=A(j+nbp))) and
BitonicMin(A,Ap,low,nbp) and BitonicMax(A,Ap,low,nbp) =>
FirstInfSecond(Ap,low,hi,nbp)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cas non courant
% où j n'existe pas en raison de
% FORALL x : x>=low and x<low+nbp => A(x)>A(x+nbp)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PROP_NOJ_0 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => BitonicList(A,low,hi,i) and (FORALL x: x>=low and
x<low+nbp => A(x)>A(x+nbp)) => (FORALL x,y:x>=low+nbp and x<=y and
y<=hi => A(x)>=A(y))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% dans ce cas on obtient deux listes bitoniques
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

BIT_NOJ_0 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => BitonicList(A,low,hi,i) and (FORALL x: x>=low and
x<low+nbp => A(x)>A(x+nbp)) and (FORALL x,y:x>=low+nbp and x<=y and
y<=hi => A(x)>=A(y)) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => DecreasingList(Ap,low,low+nbp-1) and
((EXISTS i:BitonicList(Ap,low+nbp,hi,i)) or IncreasingList(Ap,low+nbp,hi))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% les éléments de la première étant inférieurs à ceux de la seconde
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

INF_NOJ_0 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => BitonicList(A,low,hi,i) and (FORALL x: x>=low and
x<low+nbp => A(x)>A(x+nbp)) and (FORALL x,y:x>=low+nbp and x<=y and
y<=hi => A(x)>=A(y)) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => FirstInfSecond(Ap,low,hi,nbp)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cas non courant
% où j n'existe pas en raison de
% FORALL x : x>=low and x<low+nbp => A(x)<A(x+nbp)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
PROP_NOJ_1 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => BitonicList(A,low,hi,i) and (FORALL x: x>=low and
x<low+nbp => A(x)<A(x+nbp)) => (FORALL x,y:x>=low and x<=y and
y<low+nbp => A(x)<=A(y))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% dans ce cas on obtient deux listes bitoniques
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

BIT_NOJ_1 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => BitonicList(A,low,hi,i) and (FORALL x: x>=low and
x<low+nbp => A(x)<A(x+nbp)) and (FORALL x,y:x>=low and x<=y and
y<low+nbp => A(x)<=A(y)) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => IncreasingList(Ap,low,low+nbp-1) and
((EXISTS i:BitonicList(Ap,low+nbp,hi,i)) or DecreasingList(Ap,low+nbp,hi))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% les éléments de la première étant inférieurs à ceux de la seconde
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

INF_NOJ_1 : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => BitonicList(A,low,hi,i) and (FORALL x: x>=low and
x<low+nbp => A(x)<A(x+nbp)) and (FORALL x,y:x>=low and x<=y and
y<low+nbp => A(x)<=A(y)) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => FirstInfSecond(Ap,low,hi,nbp)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cas non courant % liste croissante
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
BIT_INC : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => IncreasingList(A,low,hi) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => IncreasingList(Ap,low,low+nbp-1) and
IncreasingList(Ap,low+nbp,hi)

INF_INC : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => IncreasingList(A,low,hi) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => FirstInfSecond(Ap,low,hi,nbp)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cas non courant % liste décroissante
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
BIT_DEC : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
nbp>=1 => DecreasingList(A,low,hi) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => DecreasingList(Ap,low,low+nbp-1) and
DecreasingList(Ap,low+nbp,hi)

INF_DEC : LEMMA hi>low and low>0 and nbp=nb/2 and nb=hi-low+1 and
```

```
nbp>=1 => DecreasingList(A,low,hi) and BitonicMin(A,Ap,low,nbp) and
BitonicMax(A,Ap,low,nbp) => FirstInfSecond(Ap,low,hi,nbp)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% recherche des cas non courants
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  OTHER_CASE : THEOREM
hi>low and nbp>=1
=>
  ( not (EXISTS j:j>low and j<=low+nbp-1 and
     A(j-1)<=A(j-1+nbp) and A(j)>=A(j+nbp)
  )
  <=>
   (FORALL j: j<=low or j>low+nbp-1 or A(j-1)>A(j-1+nbp) or A(j)<A(j+nbp)))

  END bitonic_sort
```

D

Quatrième annexe

Cette annexe contient le code source en C++ du tri bitonique parallèle que nous avons obtenu d'après la spécification PVS du chapitre 8. La traduction a été faite à la main.
Nous indiquons les différences qu'il existe entre la spécification et ce code :

- Nous pour chaque fonction nous avons une version pour les listes croissantes et une version pour les listes décroissantes.
- Les fonctions `bitonicsplit` ne sont pas récursives. Nous avons choisi de les implanter avec une boucle `for` pour améliorer les performances.

```
#include <iostream.h>
#include <stdlib.h>
#include <sys/time.h>


//2^24 = 16777216
//2^16 = 65536
//2^18 = 262144
//2^19 = 524288
//2^20 = 1048576
//2^21 = 2097152
//2^22 = 4194304

const long nb=1048576;

void minmaxinc(float* a,long low,long nbp)
{
  int x;
  for(x=low;x<low+nbp;x++)
  {
    float i,j;
    i=a[x];
    j=a[x+nbp];
    if(i>j) {
      a[x]=j;
      a[x+nbp]=i;
    }
```

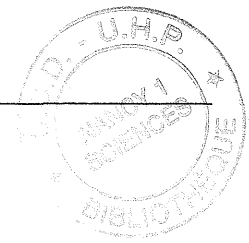
```
  }
}

void minmaxdec(float* a,long low,long nbp)
{
  int x;
  for(x=low;x<low+nbp;x++)
  {
    float i,j;
    i=a[x];
    j=a[x+nbp];
    if(i<j) {
      a[x]=j;
      a[x+nbp]=i;
    }
  }
}

void bitonicsplitinc(float* a,long low,long hi,long nb_)
{
  int i,j;
  for(i=nb_/2;i>=1;i/=2)
    for(j=0;j+i<(hi-low+1);j+=i*2)
      minmaxinc(a,low+j,i);
}

void bitonicsplitdec(float* a,int low,int hi,int nb_)
{
  int i,j;
  for(i=nb_/2;i>=1;i/=2)
    for(j=0;j+i<(hi-low+1);j+=i*2)
      minmaxdec(a,low+j,i);
}

void bitonicmerge(float* a, long low,long hi,long n_cur,long n_stop)
{
  if(n_cur<=n_stop)
  {
    long b;
    #pragma omp parallel for private(b)
    for(b=0; b<(hi-low+1)/n_cur; b++)
    {
      if (b%2==0)
        bitonicsplitinc(a,low+b*n_cur,low+(b+1)*n_cur-1,n_cur);
      else
        bitonicsplitdec(a,low+b*n_cur,low+(b+1)*n_cur-1,n_cur);
    }
    n_cur=n_cur*2;
```



```
    bitonicmerge(a,low,hi,n_cur,n_stop);
  }
}

void main( )
{
  long i;
  float *a = new float[nb];
  cout<<"generation"<<endl;
  {
  for(i=0;i<nb;i++)
    a[i]=random()/10000;
  }
  cout<<"end of generation"<<endl;

  bitonicmerge(a,0,nb-1,2,nb);

  delete []a;
}
```

Index

Bibliographie

[Abrial, 1996] J.R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[Back et Kurki-Suonio, 1988] Ralph-J. J. Back et Reino Kurki-Suonio. Distributed Co-operation with Actions Systems. *ACM Transactions on Programming Languages and Systems*, 10(4) :513–554, 1988.

[Batcher, 1968] Kenneth E. Batcher. Sorting networks and their applications. Dans *AFIPS Spring Joint Computer Conference 32*, pages 307–314, Reston, VA, 1968.

[Berche *et al.*, 1998] P.E. Berche, C. Chatelain et B. Berche. Aperiodicity-induced second-order phase transition in the 8-state potts model. *Physical Review Letters*, 80 :297, 1998.

[Bilardi et Nicolau, 1986] Gianfranco Bilardi et Alexandru Nicolau. Adaptive bitonic sorting : An optimal parallel algorithm for shared memory machines. Technical Report TR-86-769, Cornell University, 1986.

[Bolognesi et Brinksma, 1987] T. Bolognesi et E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14 :25–59, 1987.

[Chandy et Misra, 1988] K. M. Chandy et J. Misra. *Parallel Program Design A Foundation.* Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[Charpentier, 1997] Michel Charpentier. *Assitance à la répartition de systèmes réactifs.* Thèse, Institut national polytechnique de Toulouse, 1997.

[Clarke *et al.*, 1994] E. Clarke, O. Grumberg et D. Long. Verification tools for finite-state concurrent systems. Dans *A Decade of Concurrency - Reflections and Perspectives*, 803. Lecture Notes in Computer Science, 1994.

[Clarke et Emerson, 1981] E.M. Clarke et E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. Dans *Logic of Programs : Workshop*, Yorktown Heights, NY, May, 1981. Springer-Verlag. Lecture Notes in Computer Science,Vol :131.

[Clarke et Wing, 96] Edmund M. Clarke et Jeannette M. Wing. Formal Methods : State of the Art and Future Directions. Rapport technique, CMU Computer Science Technical Report, 96.

[Collard, 1995] Jean-Francois Collard. *Parallélisation automatique des programmes à contrôle dynamique.* Thèse, Université de Paris VI, 1995.

[Couturier et Méry, 1998a] Raphaël Couturier et Dominique Méry. An experiment in parallelizing an application using formal methods. Dans *International Conference on Computer Aided Verification - CAV'98, Vancouver, Canada*, rédacteurs Alan Hu et Moshe Vardi, Lecture Notes in Computer Science, Juin 1998.

[Couturier et Méry, 1998b] Raphaël Couturier et Dominique Méry. Parallelization of a Monte Carlo simulation of a spins system. Dans *Parallel and Distributed Processing Techniques and Applications - PDPTA'98, Las Vegas, USA*, rédacteur Hamid R. Arabnia, Juillet 1998.

[Crow *et al.*, 1995] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar et Mandayam Srivas. A tutorial introduction to PVS. Présenté à WIFT '95 : Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, Avril 1995.

[Darlington *et al.*, 1995] John Darlington, Yi ke Guo, Hing Wing To et Jin Yang. Parallel Skeletons for Structured Composition. Dans *Fifth ACM Conf. on Principles and Practice of Parallel Programming (PPoPP)*, pages 19–28, Santa Barbara, CA, July 1995.

[Darte *et al.*, 1996] A. Darte, F. Desprez, J.C. Mignot et Y. Robert. TransTool : A Restructuring Tool for the Parallelization of Applications Using High Performance Fortran. *Journal of the Brazilian Computer Society*, 3(2) :5–15, November 1996.

[Dijkstra, 1976] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[Dillon, 1997] Eric Dillon. *Propositions pour la maîtrise de la programmation par échange de messages.* Thèse, Université Henri Poincaré, 1997.

[Donald et Berndt, 1989] A. Donald et J. Berndt. *C-Linda reference manual.* Scientific computing associates, 1989.

[Dutertre et Stavridou, 1997] Bruno Dutertre et Victoria Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5) :267–278, Mai 1997.

[Foster *et al.*, 1992] I. Foster, R. Olson et S. Tuecke. Productive parallel programming : The PCN approach. Dans *Scientific Programming*, 1992.

[Foster, 1995] Ian Foster. *Designing and building parallel programs.* Addison Wesley, 1995.

[Galibert, 1997] Olivier Galibert. YLC, A C++ Linda system on top of PVM. Dans *4th European PVM-MPI '97, Cracow, Poland*, volume 1332 de *Lecture Notes in Computer Science*, pages 99–106. Springer-Verlag, Novembre 1997.

[Gamboa, 1997] Ruben A. Gamboa. Defthms about zip and tie : Reasoning about powerlists in ACL2. Technical Report CS-TR-97-02, The University of Texas at Austin, Department of Computer Sciences, Janvier 23 1997.

[Geist *et al.*, 1994] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek et V. Sunderam. *PVM : A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[Gelernter et Carriero, 1992] D. Gelernter et N. Carriero. Coordination language and their significance. Dans *Communications of the ACM*, volume 35 de *2*, pages 97–107, February 1992.

[Gropp *et al.*, 1994] W. Gropp, E. Lusk et A. Skjellum. *Using MPI : portable parallel programming with the message passing interface.* MIT Press, 1994.

[High Performance Fortran Forum, 1997] High Performance Fortran Forum. High Performance Fortran Language Specification, 1997. available at ftp://ftp.vcpc.univie.ac.at/HPFF/hpf-v20.ps.gz.

[Hoare, 1969] C.A.R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–583, 1969.

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1985.

[Hooman, 1997] Jozef Hooman. Program design in PVS. Dans *Workshop on Tool Support for System Development and Verification*, rédacteurs K. Berghammer, J. Peleska et B. Buth, Bremen, Germany, 1997.

[Jones, 1986] C. B. Jones. *Sytematic Software Development Using VDM.* Prentice-Hall International, 1986.

[Kellomäki, 1997] Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. Dans *Formal Methods Europe FME '97*, volume 1313 de *Lecture Notes in Computer Science*, pages 589–604, Graz, Austria, Septembre 1997. Springer-Verlag.

[Kurki-Suonio, 1998] R. Kurki-Suonio. The disco homepage. Homepage, Tampere University of Technology, Software Systems Laboratory, http://www.cs.tut.fi/laitos/DisCo/DisCo-english.fm.html, 1998.

[Ladkin *et al.*, 1996] Peter Ladkin, Leslie Lamport, Bryan Olivier et Denis Roegel. Lazy Caching in TLA. *Distributed Computing*, 1996. À paraître.

[Lamport, 1994] Leslie Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3) :872–923, May 1994.

[Lamport, 1997] Leslie Lamport. The operators of TLA+. Rapport Technique 1997-006a, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, June 1997.

[Lamport, 1998] Leslie Lamport. The module structure of TLA+. Rapport Technique 1996-002a, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, January 1998.

[Lehoucq *et al.*, 1996] R.B. Lehoucq, D.C Sorensen et C. Yang. *Arpack Users Guide*, 1996. Draft.

[Massingill, 1998] Berna L. Massingill. Experiments with program parallelization using archetypes and stepwise refinement. Dans *Lecture Notes in Computer Science*, rédacteur José Rolim, volume 1388, 1998.

[Merz, 1997] Stephan Merz. Isabelle/TLA. http://www4.informatik.tu-muenchen.de/~merz/isabelle, 1997.

[Milner, 1980] R. Milner. *A Calculus of Communicating Systems*, volume 92 de *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Misra, 1994] Jayadev Misra. Powerlist : A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6) :1737–1767, Novembre 1994.

[Open consortium, 1997a] Open consortium. OpenMP Fortran Application Program Interface, October 1997. see http://www.openmp.org/.

[Open consortium, 1997b] Open consortium. White paper : OpenMP : A Proposed Standard API for Shared Memory Programming, 1997. see http://www.openmp.org/.

[Owenson *et al.*, 1987] B. Owenson, M.A. Wilson et A. Pohorille. Cosmos – a software package for computer simulations of molecular systems. Rapport technique, NASA – Ames Research Center, Moffet Field, CA 94035-1000, 1987.

[Owicki et Gries, 1976] Susan Owicki et David Gries. An axiomatic proof technique for parallel programs i. Dans *Acta Informatica 6*, pages 319–340. Springer Verlag, 1976.

[Owre *et al.*, 1998] S. Owre, N. Shankar, J. M. Rushby et D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Septembre 1998.

[Paulson, 1994] L. C. Paulson. *Introduction to Isabelle*. Computer Laboratory, University of Cambridge, première édition, 1994.

[Paulson, 1999] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. Rapport Technique 467, University of Cambridge, 1999.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. Dans *18th ACM Symposium on the Foundations of Computer Science*, pages 46–57. ACM, 1977.

[Saïdi, 1997] Hassen Saïdi. The invariant checker : Automated deductive verification of reactive systems. Dans *Computer-Aided Verification, CAV '97*, rédacteur Orna Grumberg, volume 1254 de *Lecture Notes in Computer Science*, pages 436–439, Haifa, Israel, Juin 1997. Springer-Verlag.

[Sanders, 1991] Berverly A. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, 3 :189–205, 1991.

[Sere, 1990] Kaisa Sere. *Stepwise derivation of parallel algorithms*. PhD thesis, ÅBO AKADEMI, 1990.

[Spivey, 1987] J. M. Spivey. *Understanding Z : a specification language and its formal semantics.* Cambridge University Press, 1987.

[Sun Microsystems, 1998] Sun Microsystems, 1998. http://www.suntest.com/JavaCC/.

[Van de Velde, 1994] Eric F. Van de Velde. *Concurrent scientific computing.* Springer-Verlag, 1994.

[Vattulainen, 1994] I. Vattulainen. *News tests of random numbers for simulations in physical systems.* PhD thesis, University of Helsinki, Finland, 1994.

[White, 1992] S.R. White. Density matrix formulation for quantum renormalization groups. *Physical Review Letters*, 69 :2863, 1992.

[Zagha *et al.*, 1996] Marco Zagha, Brond Larson, Steve Turner et Marty Itzkowitz. Performance analysis using the mips r10000 performance counters. Dans *SuperComputing'96*, Pittsburgh, 1996.

Université HENRI POINCARÉ
NANCY 1

FACULTÉ DES SCIENCES

Monsieur **COUTURIER Raphaël**

DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 7 février 2000 n° 361

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY I
LE PRÉSIDENT
C. BURLET

Résumé

Le travail décrit dans cette thèse a pour but d'étudier comment on peut appliquer les méthodes formelles à la parallélisation, pour développer des programmes parallèles corrects. Comme un de nos objectifs est de travailler sur des applications en grandeur nature, nous avons, durant ce travail, collaboré avec des physiciens et chimistes de notre Université afin de paralléliser trois de leurs applications. Ces applications ont été parallélisées, sur l'Origin 2000 du Centre Charles Hermite, soit avec OpenMP, soit avec MPI, soit avec ces deux paradigmes à la fois.
Afin de prouver qu'une parallélisation basée sur une décomposition de domaines est correcte, nous avons développé une méthodologie adéquate qui demande à l'utilisateur d'abstraire son code séquentiel afin d'en obtenir une post-condition. Celle-ci nécessite d'être généralisée pour le code parallèle. Ensuite, on doit prouver que les post-conditions du code parallèle, plus la post-condition du code réalisant le collage des informations obtenues en parallèle, impliquent la post-condition du programme séquentiel. Si on ne spécifie pas la post-condition du code réalisant le collage, la preuve échoue, mais les obligations de preuves constituent la post-condition du code d'assemblage des calculs parallèles. Nous avons appliqué cette méthodologie à deux des parallélisations que nous avons effectuées.
Pour montrer que l'on peut élaborer un programme à partir d'une spécification formelle et en faire la preuve, nous avons prouvé que le tri bitonique, facilement parallélisable, est correct en utilisant PVS. Nous avons également construit un compilateur qui permet de transformer une spécification UNITY d'un programme parallèle déterministe en un programme Fortran que l'on peut exécuter sur une machine avec OpenMP.

Mots-clés: Méthodes formelles, parallélisation, preuve, OpenMP, MPI, PVS, UNITY, décomposition de domaines.

Abstract

The goal of the work described in this thesis is to study how formal methods can be applied to the parallelization of sequential programs. The use of formal methods helps to ensure that parallelization is correct. One of our objectives being to work on real applications, we have collaborated with physicists and chemists of our University to parallelize three of their simulations. These applications have been parallelized on the Origin 2000 at the Charles Hermite Centre, using either OpenMP, or MPI, or both.

To prove that domain decomposition based parallelization is correct, we have developed a method that requires the user to make an abstraction of his/her sequential code in order to obtain a postcondition. We must then generalize it for the parallel code. Consequently, we must prove that the postconditions of the parallel code together with the postcondition of the code glueing the parallel parts imply the postcondition of the sequential code. If we do not specify the postcondition of the code gathering the parallel parts then the proof fails, but proof obligations correspond to the postcondition of the code connecting the parallel parts. We have applied this method to two of our parallelizations.

In order to show that one may design programs from formal specifications, together with their proofs, we proved that the bitonic sort, which is easily parallelized, is correct using PVS. We have also built a compiler that translates UNITY specifications of deterministic parallel programs into Fortran 77 programs running on machines with OpenMP.

Keywords: Formal methods, Parallelization, Proof, OpenMP, MPI, PVS, UNITY, Domain decomposition.