Contribution à la distribution et à la synchronisation des Systèmes Multi-Agents sur les super-calculateurs

Alban Rousset

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

N° | X | X | X

THÈSE présentée par

ALBAN ROUSSET

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Contribution à la distribution et à la synchronisation des Systèmes Multi-Agents sur les super-calculateurs

Soutenue publiquement le 14/10/2016 devant le Jury composé de :

FABIEN MICHEL	Rapporteur	Maître de Conférences HDR à l'Université de Montpelier
RAYMOND NAMYST	Rapporteur	Professeur à l'Université de Bordeaux
SÉBASTIEN VARRETTE	Examinateur	Adjoint de recherche à l'Université du Luxembourg
RENÉ MANDIAU	Examinateur	Professeur à l'Université de Valenciennes et du Hainaut-Cambrésis
RAPHAËL COUTURIER	Examinateur	Professeur à l'Université de Franche-Comté
LAURENT PHILIPPE	Directeur de thèse	Professeur à l'Université de Franche-Comté
BÉNÉDICTE HERRMANN	Encadrant de thèse	Maître de Conférences à l'Université de Franche-Comté
CHRISTOPHE LANG	Encadrant de thèse	Maître de Conférences à l'Université de Franche-Comté
NICOLAS MARILLEAU	Invité	Ingénieur de Recherche à l'institut de Recherche et Développement de Bondy

Table des matières

Table des figures

Liste des tableaux

Introduction

Cadre de la thèse

Cette thèse se déroule au sein de l'équipe CARTOON du département DISC de l'institut FEMTO-ST de Besançon. Elle se situe à l'intersection des domaines des systèmes multi-agents et du parallélisme. Nous nous intéressons plus exactement à : " la distribution efficace et à la synchronisation dans les plateformes de simulations multi-agents pour leurs exécutions sur super-calculateurs".

La simulation numérique est devenue le troisième pilier de la science en tant que première étape de validation de la théorie et déterminante pour le passage à l'expérimentation. La simulation vise à virtualiser le monde réel, à en reproduire les comportements, par exemple, pour explorer son évolution dans différentes configurations ou pour comprendre comment le contrôler en l'observant. Une simulation est basée sur une représentation alternative qui est généralement simplifiée par rapport à la réalité, cette représentation se nomme modèle.

L'objectif d'un modèle est de représenter la réalité en facilitant la compréhension de celle-ci. La conception d'un modèle repose sur une étape appelée, la modélisation. Durant la modélisation, des règles sont définies sur la base de lois d'observation ou d'expériences pour reproduire la réalité. Ensuite, le modèle peut être simulé avec différents jeux de paramètres pour observer des phénomènes dans des conditions particulières.

Dans le domaine de la simulation, nous cherchons souvent à repousser les limites, c'est-à-dire à analyser des modèles plus grands et plus précis pour se rapprocher de la réalité d'un problème. L'accroissement de la précision ou de la taille des modèles simulés a un impact direct sur la quantité de calcul à réaliser. Les architectures centralisées ne sont malheureusement, plus ou pas suffisantes pour exécuter des simulations de grandes tailles. En effet, lors de l'exécution de modèles de grandes tailles, les systèmes centralisées souffrent d'un manque de mémoire ou bien de puissance de calcul. L'utilisation de ressources parallèles, c'est à dire l'utilisation de moyens de calcul haute performance (HPC : High Performance Computing) devient alors nécessaire afin de s'abstraire des limites de ressources des systèmes centralisés et ainsi d'augmenter la taille ou la précision des modèles simulés. Si la simulation sur un seul ordinateur est souvent complexe, l'exécution parallèle d'une simulation est un vrai enjeu.

Dans le monde de la simulation, il y a plusieurs manières de modéliser un système. Par exemple, la modélisation du comportement temporel d'un grand nombre de systèmes physiques repose sur des équations différentielles. La discrétisation du modèle permet de le représenter sous la forme d'un système linéaire et d'utiliser les bibliothèques parallèles existantes pour tirer parti d'un grand nombre de *nœuds* de calcul. A l'inverse, un système complexe est un système composé d'un grand nombre d'entités hétérogènes, où les interactions entre les entités créent de multiples niveaux de structure et d'organisation qui empêchent un observateur de prévoir un comportement ou sa rétroaction [1].

Les systèmes multi-agents sont souvent utilisés pour modéliser des systèmes complexes. En

1. Définition d'un système complexe dans Complex System Society

effet, les systèmes multi-agents permettent de décomposer un modèle en entités indépendantes appelées agents. Les agents reposent sur une description algorithmique simple, incluant les interactions, qui représente le comportement attendu. La dynamique des systèmes multi-agents dépend alors à la fois de la multiplicité des comportements et des interactions des agents composants la simulation. Lorsque la taille du système à représenter augmente, la simulation des systèmes multi-agents est soumise aux mêmes règles que les autres simulations. C'est-à-dire, une limitation en puissance de calcul et en mémoire qui nécessite l'utilisation d'architectures parallèles pour pallier ce problème.

Avant de continuer, il nous faut définir clairement les problématiques et questions de recherches auxquelles nous nous intéressons, c'est ce que nous faisons dans la section suivante.

Sujet et questions de recherche

Comme dit précédemment, ce travail de thèse est à la convergence de plusieurs sous-domaines de l'informatique : d'un côté le domaine multi-agents et de l'autre le domaine parallèle et distribué. Notre travail consiste à rapprocher ces deux domaines pour tenter de répondre à une question générale : est-il possible d'élaborer ou d'utiliser des techniques ou méthodes issues du domaine du parallélisme pour permettre aux systèmes multi-agents d'être parallélisés de manière efficace et ainsi bénéficier de plus de puissance de calcul et de mémoire ? Les problèmes sous-jacents étant nombreux, nous nous sommes fixés les trois questions suivantes pour nos travaux de recherche :

— **Q1** : Quelle modélisation utiliser pour efficacement distribuer une simulation multi-agents au sein d'un environnement parallèle ?

— **Q2** : Comment synchroniser et garantir la cohérence de la simulation multi-agents parallèle ?

— **Q3** : Comment gérer la communication entre les agents dans une simulation multi-agents parallèle ?

Pour répondre à ces différentes questions, nous apportons plusieurs contributions. La première est une comparaison qualitative et quantitative des plateformes multi-agents parallèles existantes. Cette comparaison nous a permis d'en déterminer les limites et d'identifier plus clairement les principales problématiques sous-jacentes à la parallélisation. Pour la seconde contribution, nous proposons une manière de modéliser les simulations multi-agents à base de *Nested Graphs*. En utilisant les graphes, cette approche facilite la distribution des simulations sur plusieurs *cœurs* et permet d'utiliser des outils et algorithmes parallèles existants. La troisième contribution concerne la synchronisation dans les simulations multi-agents parallèles. Nous nous sommes attachés à comprendre les différents niveaux de synchronisation nécessaires à l'implémentation distribuée d'un modèle, leur impact au niveau du temps d'exécution mais aussi sur les résultats. Pour la quatrième contribution, nous avons abordé le problème de la communication entre agents distribués. Nous proposons un schéma de communication qui permet aux agents de communiquer avec les agents qui s'exécutent sur n'importe quel cœurs participant à la simulation tout en respectant les contraintes de cohérence.

Plan du mémoire

Suite à cette introduction le manuscrit se décompose en deux parties. Dans cette section nous effectuons une brève présentation des chapitres qui composent ce mémoire de thèse afin d'offrir une vue d'ensemble du document et de permettre d'appréhender au mieux notre démarche.

Partie 1 : Contexte et problématiques

Le but de cette partie est d'expliciter les concepts nécessaires à la compréhension des démarches qui ont été les nôtres durant ce travail. Nous dressons ensuite un état de l'art des différents concepts que nous utilisons avant de nous positionner et de dresser un bilan des contraintes que nous rencontrons. Cette partie se décompose en deux chapitres que nous allons détailler.

Chapitre 1 : Contexte et définitions

Dans ce premier chapitre, nous introduisons le contexte de cette thèse et nous posons les définitions liées à ce contexte. Dans un premier temps nous rappelons le concept agent et les systèmes multi-agents, afin de familiariser les lecteurs avec ce domaine et les différentes notions qui le composent. Nous abordons ensuite, de manière générale, d'une part la simulation multi-agents et, d'autre part, le parallélisme. Nous dégageons alors les problématiques auxquelles nous nous intéressons au cours de cette thèse, à savoir les systèmes multi-agents parallèles.

Chapitre 2 : Les plateformes multi-agents et le parallélisme

Dans ce second chapitre, nous détaillons une première contribution qui est un comparatif détaillé de manière qualitative et quantitative des plateformes multi-agents parallèles existantes. Nous proposons des critères permettant de comparer les plateformes, aussi bien au niveau des fonctions offertes aux utilisateurs qu'au niveau du support à la parallélisation. Une comparaison des performances de ces plateformes permet également d'évaluer leur efficacité. Cette étude nous permet de mettre en évidence les problèmes sous-jacents aux systèmes multi-agents parallèles et les manques des plateformes existantes.

Partie 2 : FractalPMAS : une approche de la parallélisation des Systèmes Multi-Agents à base de *Nested Graphs*

La deuxième partie de ce mémoire présente nos travaux et les contributions que nous apportons dans les systèmes multi-agents parallèles ainsi que les outils que nous fournissons pour appuyer la démarche proposée, à savoir la plateforme multi-agent parallèle FractalPMAS.

Chapitre 3 : Distribuer les Systèmes Multi-Agents avec les *Nested Graphs*

Dans ce chapitre, nous présentons une première contribution qui est un formalisme de modélisation des systèmes multi-agents à base de graphes et plus particulièrement à base de *Nested Graphs*. Après une explication formelle de notre méthode de modélisation, nous l'appliquons, de manière graphique, à un exemple le modèle proie-prédateur. Nous montrons ensuite l'intérêt d'utiliser des graphes pour modéliser une simulation multi-agents devant être parallélisée. En effet, grâce à la représentation sous forme de graphes, nous pouvons bénéficier de la bibliothèque de partitionnement parallèle Zoltan pour distribuer efficacement la simulation sur plusieurs processus. Cette approche est validée par des expériences comparant notre plateforme aux plateformes existantes et montrent l'efficacité de la distribution.

Chapitre 4 : Les pistes de synchronisation et leurs impactes sur les modèles

Dans ce chapitre, nous présentons les problèmes liés à la synchronisation dans les systèmes multi-agents parallèles. Après avoir expliqué les différents modes de simulations, à savoir *Ghost* et *non Ghost* ainsi que les contraintes qu'ils imposent, nous définissons plusieurs politiques de synchronisation permettant de répondre aux besoins de différents types de modèles et d'interactions entre les agents. Pour évaluer l'impact de ces niveaux de synchronisation nous comparons

leurs résultats et leurs temps d'exécutions. Pour finir, nous initions une discussion sur la synchronisation dans les systèmes multi-agents parallèles.

Chapitre 5 : Gérer la communication dans les Systèmes Multi-Agents parallèles

Pour finir, dans le dernier chapitre, nous abordons le problème de la communication entre agents lorsque la plateforme est distribuée. Ces communications sont indispensables car elles servent de support aux interactions directes entre les agents. Nous proposons un schéma de communication qui permet aux agents de communiquer avec les agents exécutés sur le processus local mais aussi avec les agents exécutés sur les processus distants tout en respectant les contraintes de temps sous-jacentes à l'approche multi-agent. Après avoir expliqué l'algorithme de fonctionnement de notre schéma de communication, nous comparons les performances des communications en utilisant le modèle de référence utilisé initialement les plateformes existantes, ce qui permet d'évaluer les limites de notre proposition.

Enfin, nous dressons la conclusion du travail effectué durant cette thèse et effectuons le bilan des actions menées. Pour finir, nous proposerons différentes perspectives de travail qui permettront d'améliorer et d'enrichir les résultats obtenus.

Première partie

Contexte et problématiques

Chapitre 1

Contexte et définitions

Sommaire

La simulation numérique est devenue le troisième pilier de la science en tant que première étape de validation de la théorie et déterminante pour le passage à l'expérimentation. Elle vise à virtualiser le monde réel, à en reproduire les comportements, par exemple pour explorer son évolution dans différentes configurations ou pour comprendre comment le contrôler.

Il existe une large gamme de méthodes permettant de modéliser puis de simuler un système. La modélisation à partir d'équations différentielles est très largement utilisée dans une approche dite "Top-down" où l'ensemble des composants du système est régi par une loi de comportement unique. Elle conduit généralement à une représentation basée sur l'algèbre linéaire ou des approches de type monte-carlo. Dans le cadre de cette thèse, nous nous intéressons plus particulièrement aux systèmes complexes. Ladyman et al. dans [7] définit un système complexe par : "*Un système complexe est un système composé d'un grand nombre d'entités hétérogènes, où les interactions entre les entités créent de multiples niveaux de structure et d'organisation qui empêchent un observateur de prévoir un comportement ou sa rétroaction*". Ainsi, dans le cadre des systèmes complexes, plusieurs phénomènes peuvent être étudiés simultanément et les comportements sont souvent trop élaborés et interdépendants pour pouvoir être modélisés par une loi unique. De nombreux systèmes peuvent ainsi être sujets à des comportements qui émergent des interactions s'effectuant au niveau microscopique (niveau entité). Généralement, le niveau microscopique contrairement au niveau macroscopique (niveau global) est basé sur des interactions et des comportements simples qui se combinent et rendent le système difficile à définir au niveau macroscopique. Une simulation permet alors l'observation de phénomènes émergeant du comportement des entités. Une telle simulation conduit alors à une approche "bottom-up".

Les systèmes multi-agents sont souvent utilisés pour modéliser le comportement dynamique des systèmes complexes car ils reposent sur une description algorithmique simple d'agents qui

interagissent entre eux.

La qualité d'une simulation dépend bien souvent de la taille et de la précision du modèle. Or l'accroissement de la taille du modèle et de sa précision entraîne de fait une augmentation du nombre de calculs réalisés et rend nécessaire le recours à des exécutions parallèles, c'est à dire l'utilisation de moyens de calcul haute performance (HPC : High Performance Computing). Si la simulation sur un seul ordinateur est souvent complexe, l'exécution parallèle d'une simulation est un vrai enjeu.

Le contexte général de cette thèse est celui de l'exécution parallèle de systèmes multi-agents. Nous allons dans un premier temps, nous intéresser spécifiquement à tout ce qui compose les systèmes et simulations multi-agents. Puis dans un deuxième temps, nous abordons les systèmes parallèles.

1.1 Les Systèmes Multi-Agents

Un Système Multi-Agents (SMA) est composé de plusieurs agents spécialisés ou dans le but de répondre à un problème posé. Il est donc nécessaire de définir le concept d'agent afin de pouvoir comprendre celui d'un système multi-agents. Une fois le système multi-agents expliqué, nous aborderons l'animation d'un modèle agent à l'aide des plateformes multi-agents. Pour finir, plusieurs modèles agents classiques sont présentés.

1.1.1 Agent

Le concept d'agent a été l'objet de nombreuses études depuis plusieurs décennies et dans différents domaines. Il est non seulement utilisé dans les systèmes à base de connaissances, la robotique et d'autres domaines de l'intelligence artificielle, mais aussi dans les domaines comme la psychologie [8, 9]. L'une des première définition communément admise du concept d'agent est due à Ferber [10] :

"*Un agent est une entité réelle ou virtuelle, évoluant dans un environnement, capable de le percevoir et d'agir dessus, qui peut communiquer avec d'autres agents, qui exhibe un comportement autonome, lequel peut être vu comme la conséquence de ses connaissances, de ses interactions avec d'autres agents et des buts qu'il poursuit.*"

A partir de cette définition, nous pouvons mettre en avant différentes capacités des agents telles que l'action, la perception et la communication. Le comportement des agents est dirigé par un ou plusieurs objectifs individuels. Selon la définition de Ferber [10], les agents sont destinés à être embarqués dans un environnement au sein duquel ils devront effectuer une tâche en commun. Ainsi le comportement d'un agent peut être décomposé en trois grandes phases (Figure 1.1) [1] : la perception, la délibération et l'action. Durant l'exécution du comportement d'un agent, ces trois phases sont exécutées de manière cyclique jusqu'à la fin de la simulation.

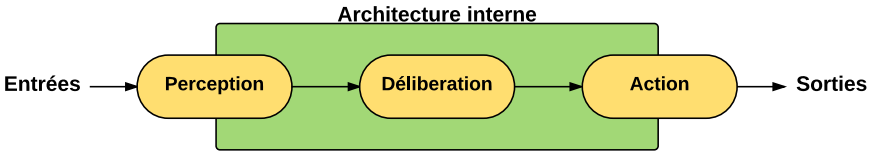


FIGURE 1.1 – Les trois phases du comportement d'un agent [1]

Ce cycle permet à un agent de percevoir son environnement, de prendre une décision en fonction de ce qu'il a perçu et d'agir. Suivant le type d'agent la phase de délibération peut être

inexistante. En effet, à l'intérieur du concept agent les auteurs Ferber et al. et Wooldridge et al. [10, 1] distinguent plusieurs types d'agents.

Les différents types d'agents

Un **agent réactif** se caractérise par des comportements simples qui peuvent être perçus comme des réactions à des stimulis provenant des autres agents ou encore de l'environnement. Ainsi le comportement d'un agent réactif est composé de deux phases : la perception et l'action. La phase de délibération est absente pour ce type d'agent. De plus, un agent réactif ne possède pas de mémoire, ni de représentation de l'environnement dans lequel il évolue. Les actions que l'agent entreprend dépendent directement de ce qu'il perçoit dans son champ de vision. Les agents réactifs sont très largement utilisés, pour faire émerger des comportements en se basant sur la synergie des agents, c'est à dire sur l'émergence d'un comportement collectif complexe basé sur une multitude d'actions simples.

Dans la majorité des cas les agents réactifs permettent de résoudre des problèmes complexes basés sur l'émergence d'un comportement global. Cependant, dans certains cas les comportements réactifs ne sont pas suffisants ou pas adaptés pour représenter le phénomène souhaité. Un autre type d'agent appelé agent cognitif peut donc être utilisé.

Un **agent cognitif**, intentionnel ou délibératif quant à lui se caractérise par une mémoire et une base de connaissances qui lui permet d'avoir une représentation (partielle ou non) de l'environnement dans lequel lui et les autres agents évoluent. Les agents cognitifs sont généralement couplés à un moteur d'inférence qui leur permet d'avoir des comportements et de prendre des décisions évoluées. C'est à dire qu'ils possèdent la capacité de délibérer pour produire des plans d'actions. Les comportements et les interactions entre agents sont organisés selon une architecture. L'une des architectures les plus populaire est l'architecture BDI (Believes, Desires, intensions) [11]. Cette architecture repose sur une idée simple : l'accomplissement d'un but (Desire) est le fruit de la réalisation d'objectifs intermédiaires (Intentions) qui sont identifiés par une analyse des croyances (Belief) de l'agent sur son environnement.

La principale différence entre les agents réactifs et les agents cognitifs est la capacité de délibération. Cette capacité de délibération est absente chez les agents réactifs. A l'aide de leur mémoire, les agents cognitifs peuvent mémoriser des situations, les analyser et les planifier ou prévoir des actions.

En résumé, les agents réactifs répondent aux stimuli et les agents cognitifs possèdent une représentation de l'environnement et sont capables de planifier leurs actions. Il est important de noter que cette catégorisation des agents n'est pas seulement limitée à ces deux extrêmes. Dans les faits, les agents sont souvent conçus au cas par cas afin de simuler au mieux des péhonmènes complexes. C'est la raison pour laquelle il est possible de combiner ces deux types d'agents pour obtenir ce que l'on appelle les agents hybrides [12, 13]. La Figure 1.2 représente schématiquement les différents types d'agents.

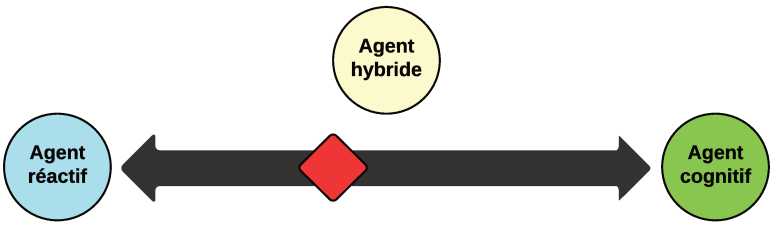


FIGURE 1.2 – Représentation schématique des différents types d'agents

Les **agents hybrides** se composent donc d'un mélange de comportements d'agents réactifs et de comportements d'agents cognitifs. En fonction des besoins nécessaires pour modéliser le phénomène souhaité, il est possible de favoriser plus un type d'agent qu'un autre. A l'heure actuelle, il existe de plus en plus d'architectures pour agents hybrides, l'une des plus connues est InteRRap [14].

1.1.2 Systèmes Multi-Agents

Un système multi-agents n'est pas uniquement formé d'un ensemble d'agents. Selon Ferber [10], un système multi-agents se compose des éléments suivants :

— un environnement E, c'est à dire un espace disposant d'une métrique ;

— un ensemble d'objets O, ces objets sont situés, c'est à dire qu'à un moment donné il est possible d'associer à tout objet une position dans E. Ces objets sont passifs, c'est à dire qu'ils peuvent êtres perçus, créés, détruits et modifiés par les agents ;

— un ensemble A d'agents, qui sont des objets particuliers ($A \subseteq O$), lesquels représentent les entités actives du système ;

— un ensemble de relations R qui unissent des objets (et donc des agents) entre eux ;

— un ensemble d'opérations Op permettant aux agents de A de percevoir, produire, consommer, transformer et manipuler des objets de O ;

— des opérateurs chargés de représenter l'application de ces opérations et la réaction du monde à cette tentative de modification, qu'on appellera les lois de l'Univers.

On peut distinguer deux grandes catégories de SMA : les simulations multi-agents et les applications à base d'agents. Les simulations multi-agents impliquent la prise en compte d'une problématique organisationnelle et collective des agents pour résoudre un problème. Ainsi, une simulation multi-agents est un système composé d'un ensemble d'agents autonomes qui évoluent dans un même espace virtuel avec pour but de fournir une fonction collective par l'intermédiaire de l'effet de synergie entre les agents. Les simulations multi-agents permettent de reproduire un phénomène naturel afin de pouvoir l'étudier. Les applications agents sont des applications classiques qui embarquent des agents en leur sein.

Les auteurs [15, 16, 17] distinguent quatre grandes catégories d'environnements en fonction de leur nature et de leur dynamique :

— l'environnement accessible (par opposition à inaccessible"). Il est possible d'obtenir une information complète, exacte et à jour sur l'état de l'environnement. A l'inverse dans un environnement inaccessible, il est impossible de tout connaitre à propos de l'environnement, seulement des connaissances partielles.

— l'environnement continu (par opposition à "discret"). Le nombre d'actions et de perceptions possible dans ce type d'environnement est infini. A l'inverse, dans un environnement discret, le nombre d'actions ou de perceptions peut être limité.

— l'environnement déterministe (par opposition à "non déterministe"). Dans ce type d'environnement, une action a un effet unique et certain. C'est à dire qu'il n'y a pas d'incertitude sur les effets d'une action sur l'état de l'environnement. A l'inverse, dans un environnement indéterminisme, une action n'a pas un effet unique garanti.

— l'environnement dynamique (par opposition à "statique"). L'état d'un environnement dynamique est entièrement dépendant des actions qui s'effectuent dans cet environnement. Il est donc difficile de prédire les changements. En revanche, dans un environnement statique, le système ne peut pas changer sans que le système agisse.

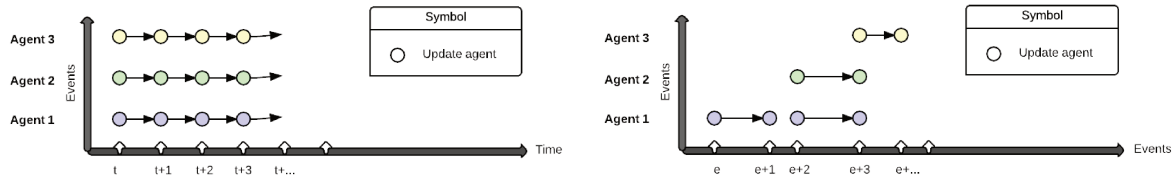
Une simulation multi-agents peut être décomposée en deux étapes : la modélisation et l'animation du modèle. La modélisation se définit classiquement comme la création d'un modèle représentant de manière simplifiée ou abstraite, une chose réelle déjà existante (objet, phénomène, etc.), en vue de la comprendre. Dans le cadre des SMA la modélisation consiste à créer un modèle représentant un phénomène sous forme d'agents. Le modèle agent une fois créé est animé par une plateforme multi-agents.

1.1.3 Animation d'un modèle multi-agents

L'animation d'un modèle multi-agents est réalisée à l'aide d'un ordonnanceur. L'ordonnanceur gère le cycle de vie des agents (création et suppression dynamique d'agents), la perception des agents, la communication entre agents et avec l'environnement. On distingue deux principales approches d'animation de modèle multi-agents [18, 19] : la simulation animée par les pas de temps (time-driven) et la simulation animée par les événements (event-driven).

Les simulations par pas de temps traduisent le déroulement chronologique du système sur une échelle de temps régulière, c'est à dire que l'ensemble des agents évoluent d'un pas de temps à un autre. Un pas de temps peut prendre différentes unités : secondes, minutes, jours, etc. Au cours d'un pas de temps, l'ordonnanceur anime à tour de rôle chaque agent de la simulation en exécutant leur comportement (Figure 1.3-a).

Dans une simulation par événements, les événements traduisent la dynamique du système. Les agents produisent des événements qui doivent être exécutés à une date donnée (Figure 1.3-b). L'ordonnanceur exécute les événements en fonction de leur date de traitement.



(a) Simulation par pas de temps

(b) Simulation par évènements

FIGURE 1.3 – Les deux types d'animation de simulations discrètes pour les simulations multi-agents

On peut remarquer qu'une simulation par pas de temps peut être développée à l'aide d'ordonnanceur par événements. Il faut alors attribuer une périodicité à chaque événement.

Le choix du type d'animation de la simulation est généralement orienté par le système que nous souhaitons modéliser ou le phénomène que nous souhaitons observer. Par exemple, si nous souhaitons modéliser le comportement de personnes dans une ville, il convient d'utiliser la simulation par pas de temps afin d'observer l'évolution au cours du temps de chaque agent et de l'ensemble de la simulation. En revanche, si nous voulons modéliser le comportement de la bourse ou encore d'incidents dans un réseau, la simulation par événements sera privilégiée car ce que nous souhaitons observer est le résultat de l'exécution des événements en ignorant les étapes intermédiaires.

La notion de temps est très présente dans les simulations multi-agents. On distingue habituellement deux types de temps : le temps simulé et le temps réel. Le temps simulé fait référence au temps qui se déroule dans la simulation tandis que le temps réel fait référence au temps mis par la simulation pour s'exécuter. Ainsi, le temps simulé par la simulation peut être beaucoup plus grand que le temps réel et vice versa.

1.1.4 Les plateformes multi-agents

L'intérêt croissant pour la modélisation à base d'agents a été favorisé par le développement de plateformes accessibles à des non informaticiens. En effet, une plateforme multi-agents ne propose pas uniquement l'animation d'un modèle agent. Elle fournit une multitude de fonctionnalités permettant de simplifier la définition d'un modèle agent, d'animer le modèle et d'analyser les résultats de l'animation par l'intermédiaire de graphiques et de courbes. Tous ces mécanismes sont essentiels au bon déroulement d'une simulation multi-agents et complexes à mettre en œuvre.

Les auteurs Banos et al. distinguent dans [20] trois catégories de plateformes en fonction du type de langage qu'elles utilisent pour définir les modèles.

La première catégorie de plateformes est principalement destinée aux informaticiens car la définition d'un modèle s'effectue par l'intermédiaire d'un langage de programmation générique tel que Java, C++, python, etc... Ces plateformes sont généralement adaptées au développement de gros modèles. Parmi les plateformes les plus connues, nous pouvons citer Madkit [21], Mason [22], Cormas [23] ou encore Repast [24].

Destinée à un plus large public, la deuxième catégorie de plateformes utilise un langage de modélisation dédié. Elles sont plus simples d'utilisation et nécessitent moins de connaissances algorithmiques. Les plateformes les plus connues de cette catégorie sont Netlogo [25] et Gama [26].

La troisième catégorie de plateformes ne nécessite pas ou très peu de connaissances algorithmiques. La définition des modèles s'effectue à l'aide d'un langage graphique. Ces langages graphiques ne permettent pas de réaliser des modèles complexes. Ainsi, ces plateformes n'offrent pas autant de richesses que les autres catégories de plateformes. Parmi ces plateformes les plus connues sont StarLogo [27] et MAGeo [28].

Il est important de noter que certaines plateformes telles que Gama [26], Cormas [23] ou encore Repast [24] proposent plusieurs types de langage pour définir un modèle. Les plateformes Gama et Cormas proposent des outils graphiques tandis que la plateforme Repast propose le langage Java, Relogo (une implémentation du langage de la plateforme logo dans un modèle Repast) ainsi que des outils graphiques.

Les articles [29, 30, 31, 32] établissent la comparaison de plateformes multi-agents.

Dans les systèmes multi-agents il existe un certain nombre de modèles agents qui sont reconnus par la communauté comme des modèles classiques. Nous avons utilisé ces modèles pour illustrer certains concepts ou effectuer des expérimentations afin de valider nos contributions. Nous les détaillons dans la section suivante.

1.1.5 Modèles agent classique

Nous expliquons le fonctionnement de chacun de ces modèles dans la suite de cette section.

Flocking [33]

Objectif du modèle

Le modèle Flocking ou troupeau est inspiré du modèle *boids* de Reynolds [34]. Ce modèle permet de simuler le vol d'une nuée d'oiseaux afin d'en étudier le comportement. La nuée d'oiseaux n'est ni créée, ni dirigée par des agents chefs.

Agents, environnements et variables d'états

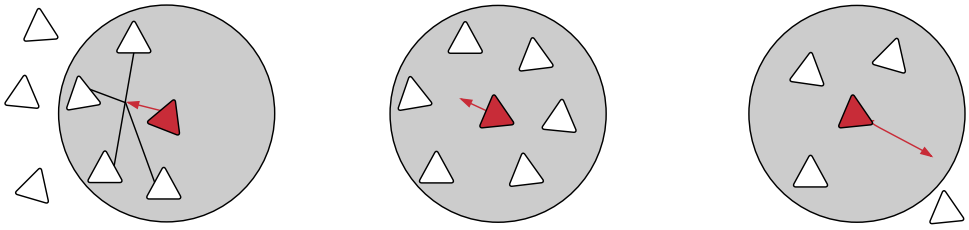
Le modèle est composé d'un seul type d'agents : les oiseaux. Ces agents sont de type réactif, c'est à dire qu'ils n'ont pas de phase de délibération. Les oiseaux ne possèdent qu'une connais-

sance partielle de l'environnement dans lequel ils évoluent et ne possèdent pas de mémoire ni de comportements évolués. Ces agents sont caractérisés par une position dans l'espace à l'aide de coordonnées cartésiennes. Chaque agent oiseau se définit à l'aide des trois comportements suivants :

— la cohésion (Figure 1.4(a)) : un oiseau a tendance à se déplacer vers les autres oiseaux proches (à moins que les autres oiseaux ne soient assez proches).

— l'alignement (Figure 1.4(b)) : un oiseau a tendance à tourner pour se déplacer dans la même direction que les oiseaux voisins.

— la séparation (Figure 1.4(c)) : un oiseau a tendance à tourner pour éviter un autre oiseau qui est trop près.



(a) Cohésion (b) Alignement (c) Séparation

FIGURE 1.4 – Description des trois comportements composant un agent oiseau

A chaque pas de temps, un agent oiseau exécute ces trois comportements qui permettent de calculer la nouvelle position de l'oiseau en fonction de la position des oiseaux qui composent son voisinage.

L'environnement dans lequel évoluent et se situent les agents oiseaux peut être soit en deux dimensions sous forme de grille, soit en trois dimensions sous forme de cube.

Plusieurs variables sont définies comme paramètres du modèle : la taille de l'environnement, la distance maximale qu'un oiseau peut avancer par pas de temps, la durée d'un pas de temps, le taux de cohésion, d'alignement et de séparation.

Virus [35]

Objectif du modèle

Le modèle Virus permet de simuler la transmission et le maintien d'un virus dans une population humaine. Ce modèle repose sur un certain nombre de facteurs pouvant influer la survie d'un virus transmis directement au sein d'une population [36].

Agents, environnements et variables d'états

Le modèle est composé d'un seul type d'agents : les personnes. Ces agents sont de type réactifs, c'est à dire qu'ils n'ont pas de phase de délibération. Les personnes ne possèdent qu'une connaissance partielle de l'environnement dans lequel ils évoluent et ne possèdent pas de mémoire ni même de comportements évolués. Ces agents sont caractérisés par une position dans l'espace à l'aide de coordonnées cartésiennes. Chaque agent se définit à l'aide des cinq comportements suivants :

— le vieillissement : une personne vieillit jusqu'à ce qu'elle meurt.

— le déplacement : une personne se déplace de manière aléatoire sur l'environnement.

— l'infection : une personne infectée par le virus, peut contaminer les personnes qui se situent dans son voisinage.

— la récupération : une personne infectée par le virus peut (selon une probabilité) devenir immunisée. Ainsi, la personne concernée ne sera plus porteuse du virus.

— la reproduction : une personne non contaminée par le virus peut (selon une probabilité) se reproduire. Ce comportement permet de renouveler la population.

A chaque pas de temps, un agent personne exécute ces cinq comportements qui permettent de faire évoluer le virus au sein de la population de personnes.

L'environnement dans lequel évoluent et se situent les agents personnes est une grille en deux dimensions.

Plusieurs variables sont définies comme paramètres du modèle : la capacité de transport du virus (combien de personnes peuvent être dans le monde à un moment donné), la durée de vie des personnes, le taux de natalité, le taux de reproduction et le nombre de personnes porteuses du virus à l'initialisation du modèle.

Proie-prédateur [37]

Objectif du modèle

Le modèle proie-prédateur explore la stabilité des écosystèmes. Dans un tel modèle, nous distinguons deux variantes : l'une appelée instable si elle a tendance à entraîner l'extinction d'une ou plusieurs espèces concernées et l'autre appelée stable si elle tend à se maintenir dans le temps, malgré les fluctuations de la taille de la population.

Nous prenons comme exemple de modèle prédateur-proie celui des loups et des moutons.

Dans la variante instable du modèle, les loups et les moutons errent au hasard dans le paysage, les loups recherchent des moutons (proie). Chaque étape coûte de l'énergie aux loups et aux moutons, quand ils n'ont plus d'énergie ils meurent. Seuls les loups peuvent reconstituer leur énergie en mangeant des moutons. Pour permettre à la population de continuer à perdurer, chaque loup et chaque mouton a une probabilité fixe de se reproduire à chaque pas de temps.

La variante stable du modèle introduit un nouveau type d'agent dans le modèle : l'herbe. Dans cette variante, le comportement des loups est identique à celui de la variante instable. Par contre, les moutons peuvent manger de l'herbe (proie) afin de maintenir leur énergie. Une fois que l'herbe est mangée, elle disparaît et repousse après un temps fixé.

Agents, environnements et variables d'états

Le modèle est composé dans sa première variante, de deux types d'agents : les loups (prédateurs) et les moutons (proies). En revanche dans sa deuxième variante, le modèle est composé de trois types d'agents : les loups (prédateurs), les moutons (proies/prédateurs) et l'herbe (proie). Ces agents sont de type réactifs, c'est à dire qu'ils n'ont pas de phase de délibération. Les trois types d'agents ne possèdent qu'une connaissance partielle de l'environnement dans lequel ils évoluent et ne possèdent pas de mémoire ni même de comportements évolués. Ces agents sont caractérisés par une position dans l'espace à l'aide de coordonnées cartésiennes et par une énergie qui représente pour les loups et les moutons leur durée de vie, l'herbe, elle, étant présente ou non. Chaque agent loup et mouton se définit à l'aide des quatre comportements suivants :

— le déplacement : les loups et les moutons se déplacent aléatoirement sur l'environnement. Lors d'un déplacement ils vieillissent.

— la nourriture : les loups et les moutons se nourrissent de proies si elles se situent dans leurs champs de perception. Cela a pour but d'augmenter leur durée de vie.

— le décès : les loups et les moutons meurent s'ils ont atteint leur durée de vie.

— la reproduction : les loups et les moutons peuvent (selon une probabilité) se reproduire. Ce comportement permet de renouveler la population.

A chaque pas de temps, les agents loups et moutons exécutent les quatre comportements précédents dans l'ordre énoncé.

Un agent herbe apparaît tous les n pas de temps où n est un nombre tiré aléatoirement entre 0 et une borne maximale fixée comme paramètres du modèle.

L'environnement dans lequel évoluent et se situent les agents est une grille en deux dimensions.

Plusieurs variables sont définies comme paramètres du modèle : la taille de l'environnement, le nombre de loups, le nombre de proies, le taux de natalité, le taux de reproduction, le gain de vie lorsque qu'un prédateur mange une proie et le temps de croissance maximal des agents herbe.

Un modèle d'évaluation des SMA parallèles

Aucun des modèles classiques ne regroupe toutes les fonctionnalités que nous souhaitons évaluer dans les plateformes SMA parallèles. Pour cette raison, nous avons défini un modèle d'évaluation des SMA parallèles

Objectif du modèle

Ce modèle repose sur trois propriétés importantes d'un agent : la perception, la communication entre les agents et avec l'environnement et la mobilité. Nous avons choisi ces trois propriétés car elles illustrent bien la définition d'un agent donnée par Ferber à la Section 1.1.1 et permettent de tester les fonctionnalités de base qu'un SMA parallèle doit fournir.

Dans ce modèle, les agents se déplacent dans un environnement qui est représenté par une grille. Un agent interagit et communique avec les autres agents qui se trouvent à l'intérieur de son champs de perception.

Agents, environnements et variables d'états

Le modèle est composé d'un seul type d'agents : les personnes. Ces agents sont de type réactifs, c'est à dire qu'ils n'ont pas de phase de délibération. Les personnes ne possèdent qu'une connaissance partielle de l'environnement dans lequel elles évoluent et ne possèdent pas de mémoire ni même de comportements évolués. Ces agents sont caractérisés par une position dans l'espace à l'aide de coordonnées cartésiennes. Chaque agent se définit à l'aide des trois comportements suivants :

— le déplacement (Walk) : une personne se déplace aléatoirement sur l'environnement. Comme les personnes se déplacent à travers l'environnement, elles découvrent d'autres personnes et d'autres parties de l'environnement. Ce comportement est utilisé pour tester la mobilité et la perception des personnes ainsi que leurs interactions et communications avec l'environnement.

— l'interaction (Interact) : une personne envoie des messages à toutes les personnes qui se trouvent dans son champ de perception. Un message est composé de l'identifiant de l'agent personne émetteur et d'une valeur entière. Ce comportement a pour but de simuler les communications entre les personnes et d'évaluer le support de communication des plateformes.

— le calcul (Compute) : consiste à simuler une charge de travail générée par l'exécution des algorithmes internes de l'agent. Ce comportement calcule une Transformation de Fourier Discrète (DFT) à une dimension [38].

Chaque agent personne exécute les trois comportements précédents à chaque pas de temps.

L'environnement dans lequel évoluent et se situent les agents est une grille en deux dimensions.

Plusieurs variables sont définies comme paramètres du modèle : la taille de l'environnement, le nombre de personnes, la taille du calcul de DFT à effectuer ainsi qu'une variable permettant d'activer ou non les communications.

1.2 Les systèmes parallèles

A l'heure où les calculs sont de plus en plus gourmands en mémoire et de plus en plus précis, le recours aux architectures parallèles est devenu une nécessité. Ces dernières années, les architectures parallèles sont devenues le paradigme dominant comme le montre l'augmentation des capacités offertes par les plus puissants super-calculateurs mondiaux du TOP500 [1]. Cela s'explique par la stagnation de l'augmentation des fréquences des processeurs avec l'arrivée des processeurs multi-cœurs, c'est à dire des processeurs capables de traiter plusieurs instructions de manière simultanée.

La parallélisation est un moyen d'accélérer, la vitesse d'exécution d'un programme ou du traitement de données, en les répartissant sur plusieurs machines ou processeurs. L'objectif du parallélisme en informatique est multiple. Les systèmes parallèles permettent, d'une part, d'avoir accès à davantage de mémoire ce qui permet de simuler des modèles de plus grande taille et, d'autre part, d'avoir accès à plus de puissance de calcul ce qui permet de réduire le temps d'exécution d'un programme. Nous définissons donc le parallélisme de la manière suivante : le parallélisme consiste à mettre en œuvre des architectures et des modèles de programmation capables de traiter des informations de manière simultanée dans le but de réduire le temps de calcul et de s'abstraire des limitations de mémoire et de puissance de calcul.

Dans la littérature, on distingue deux types d'architectures, les architectures distribuées et les architectures parallèles. La distinction entre les deux appellations n'est pas toujours clairement définie ou reconnue, les deux termes pouvant être utilisés dans certains cas. Il est alors indispensable de préciser l'utilisation que nous en avons dans ce travail.

Dans la suite de cette section, nous précisons donc les différents types d'architectures (parallèles et distribuées), puis nous détaillons les modèles de programmation associés aux architectures parallèles. Ensuite, nous décrivons les métriques qui permettent de quantifier la qualité d'une parallélisation. Pour finir, nous nous intéressons aux contraintes imposées par les systèmes parallèles, et plus particulièrement les contraintes des super-calculateurs, ainsi qu'aux limites des applications parallèles.

1.2.1 Architecture distribuée

Une architecture distribuée se définit comme un environnement informatique où toutes les ressources ne se trouvent pas au même endroit géographique ou pas sur la même machine. Généralement, les architectures distribuées sont associées aux grilles de calcul. Une grille de calcul est un ensemble de ressources informatiques hétérogènes (ordinateurs, serveurs, clusters, ...), organisées en grappe, dont l'objectif est de mettre à disposition des scientifiques une puissance de calcul distribuée. Les grilles de calculs sont délocalisées dans la mesure où les *nœuds* d'une même

1. `https://www.top500.org/`

grappe sont souvent physiquement positionnés dans plusieurs lieux géographiques différents. Les ressources peuvent être hétérogènes dans leurs caractéristiques techniques, leur système d'exploitation, leur système de gestion et de soumission des jobs. Il n'y a pas une unité commune avec un moniteur de contrôle qui contrôle les ressources.

Ces architectures sont répandues, elles peuvent elles-mêmes être composées d'un ensemble d'architectures parallèles inter-connectées que nous détaillons dans la section suivante.

1.2.2 Architecture parallèle

On définit une architecture parallèle comme un environnement informatique où toutes les ressources se trouvent au même endroit géographique et sur la même machine. Les architectures parallèles sont souvent associées aux termes clusters ou super-calculateurs. Dans la littérature, on distingue deux types d'architectures parallèles, différenciées par leur type de mémoire : les architectures parallèles à mémoire partagée et celles à mémoire distribuée.

Architecture parallèle à mémoire partagée

Dans une architecture parallèle à mémoire partagée (ou *shared memory*), tous les processeurs accèdent à toute la mémoire globale avec un même espace d'adressage comme le présente la Figure 1.5. Chaque processeurs (*CPU*) exécute indépendamment des autres des processus, mais les modifications effectuées par un processeurs à un emplacement mémoire sont visibles par tous les autres. Cependant les processeurs disposent également de leur propre mémoire locale (cache).

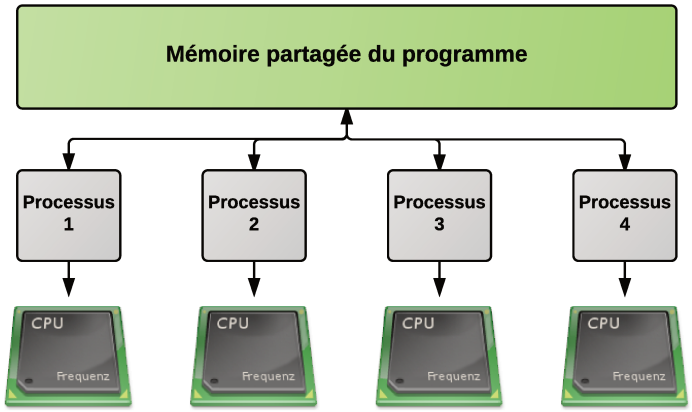


FIGURE 1.5 – Architecture parallèle à mémoire partagée

Grâce à cette architecture, le travail du programmeur est facilité car l'espace mémoire est unique pour tous les processeurs. De plus, le partage de données est rapide entre les processeurs. En revanche, il y a un réel manque d'extensibilité avec ce type d'architecture car lorsque l'on augmente le nombre de processeurs cela accroît de fait le nombre d'accès à la mémoire partagée. Un autre inconvénient est la gestion des synchronisations lors de la programmation de l'application afin de gérer les accès concurrents à la mémoire globale. Pour finir, il est très coûteux de construire des architectures à mémoire partagée qui possèdent un grand nombre de processeurs.

Pour pouvoir paralléliser un programme pour une architecture parallèle à mémoire partagée, il est nécessaire d'utiliser un écosystème logiciel ou modèle de programmation adapté.

La programmation en mémoire partagée

Le modèle de programmation standard pour les architectures parallèles à mémoire partagée est OpenMP [39]. OpenMP est disponible pour les langages de programmation C, C++ et Fortran. Il permet d'annoter à l'aide de *pragma* un code source existant afin de paralléliser un programme séquentiel, c'est à dire s'exécutant normalement sur un seul processus. A l'aide de directives, OpenMP permet entre autre de définir les régions parallèles du programme (boucles, sections parallèles), la manière dont s'effectue le partage des tâches, les points de synchronisation et permet aussi de gérer le partage des données (privées, partagées). Avec le standard OpenMP, les communications entre les processus se font par lectures et écritures dans la mémoire partagée. Pour finir, il est possible avec OpenMP de configurer le nombre de processus à utiliser.

Architecture parallèle à mémoire distribuée

A la différence des architecture à mémoire partagée, dans une architecture parallèle à mémoire distribuée chaque processeurs possède sa propre mémoire locale, il n'y a aucune notion d'espace mémoire global entre tous les processeurs comme le montre la Figure 1.6. Les processeurs (*CPU*) exécutent indépendamment les uns des autres des processus. Une modification en mémoire locale n'a aucun effet sur la mémoire des autres processeurs. Dans le cas où un processeur a besoin d'une donnée située dans la mémoire d'un autre processeur, c'est au programmeur de définir explicitement une communication. Pour finir, les réseaux d'interconnexion peuvent être divers avec des niveaux de performance très variables.

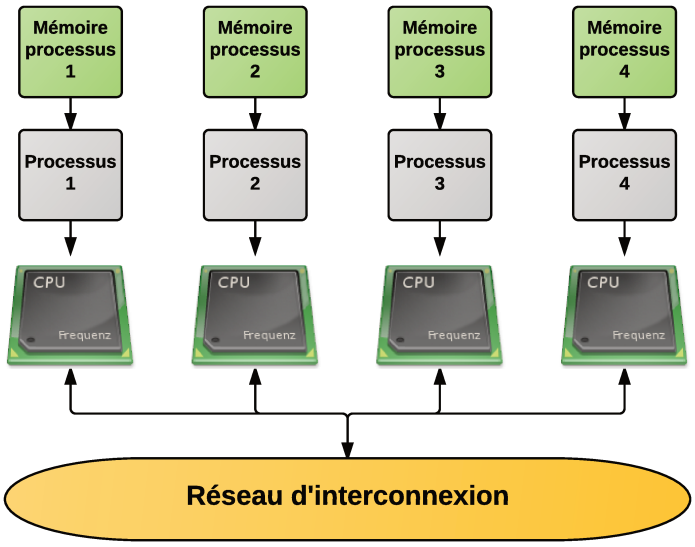


FIGURE 1.6 – Architecture parallèle à mémoire distribuée

L'avantage de cette architecture à mémoire distribuée est que l'on peut augmenter théoriquement le nombre de processeurs ainsi que la mémoire de manière infinie. L'accès à la mémoire locale est rapide sur chaque processeur et sans surcoût de gestion de cohérence. En revanche, ce type d'architecture oblige le programmeur à gérer toutes les communications entre les processeurs. Parfois, il peut être difficile d'adapter une structure de données, basée sur une mémoire globale, à cette organisation physique de la mémoire. Pour finir, les temps d'accès mémoire non locaux peuvent être élevés.

Généralement, les architectures parallèles à mémoire distribuée sont associées aux super-calculateurs CPU ou au calcul haute performance (HPC). On définit un super-calculateur CPU comme est un ensemble "d'ordinateurs" homogènes indépendants appelés *nœuds*. Les *nœuds* sont généralement localisés et organisés en grappe [40]. La Figure 1.7 représente la composition d'un *nœud* d'un super-calculateur CPU. Un super-calculateur se compose d'une multitude de *nœuds*, eux-mêmes composés de plusieurs *processeurs* multi-cœurs pouvant exécuter chacun des *processus*.

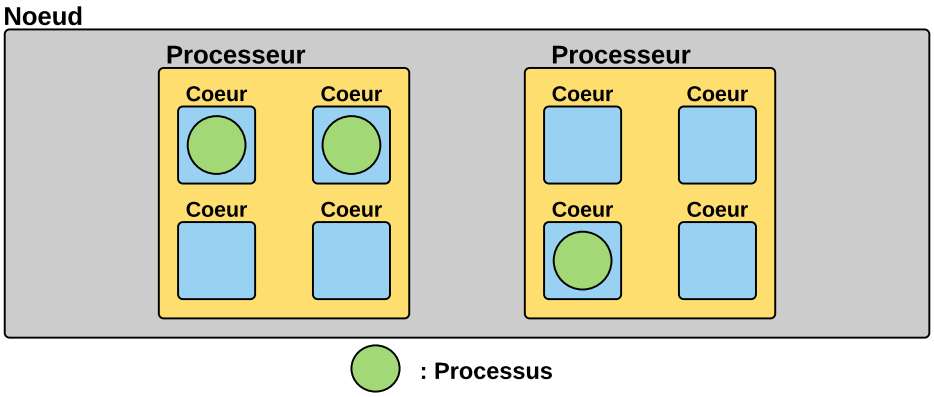


FIGURE 1.7 – Architecture d'un *nœud* d'un super-calculateur CPU

Chaque super-calculateur ou cluster utilise généralement plusieurs dizaines de *nœuds* composés de plusieurs processeurs connectés entre eux par une connexion à très haut débit et à faible latence (par exemple Infiniband). L'avantage est de créer un calculateur comme un seul ordinateur ayant plus de performances (puissance du processeur, taille de l'espace de stockage, quantité de mémoire vive, etc.). Pour rappel et à titre informatif, un classement des clusters les plus puissants au monde est proposé à l'adresse suivante `http://top500.org`.

La programmation en mémoire distribuée

Le modèle de programmation standard pour les architectures parallèles à mémoire distribuée est Message Passing Interface (MPI) [41]. MPI est une norme regroupant un ensemble de fonctions de communication entre processus reposant sur l'échange de messages. MPI est disponible pour les langages de programmation C, C++ et Fortran. L'objectif des fonctions de communication de MPI est de proposer de bonnes performances d'exécution aussi bien sur une même machine qu'entre des machines distantes.

Lors de l'exécution MPI, un numéro de rang, *Rank*, est associé à chaque processus participant. Ce numéro de rang est indépendant de la localisation physique du processus et permet à chaque processus d'être identifiable et atteignable avec les primitives de communication.

Pour finir, MPI permet deux grandes catégories de communication : les communications point-à-point et les communications collectives (de groupe).

— Les communications point-à-point : permettent de communiquer entre un unique émetteur et un destinataire parmi le groupe de processus.

— Les communications collectives : permettent d'impliquer une partie où tous les processus du groupe pour réaliser un traitement, par exemple la diffusion d'un résultat à tous les processus (*Broadcast*).

Il existe de nombreuses implémentations du standard MPI. On peut citer par exemple, les implémentations MPICH et OpenMPI qui sont libres tandis que d'autres sont propriétaires comme IntelMPI proposant des optimisations pour les processeurs Intel.

Les approches de synchronisation

La programmation en mémoire distribuée contraint le développeur à synchroniser explicitement certaines portions de l'application pour différentes raisons. En effet, la distribution de l'application sur différents processeurs possédant leur propre mémoire, c'est-à-dire autonome, implique des problèmes de coordination entre éléments distants ainsi que des problèmes d'accès aux ressources. Il existe deux grandes approches de synchronisation dans les simulations distribuées, l'approche conservative et l'approche optimiste.

L'approche conservative (ou pessimiste) est dominante dans les systèmes à mémoire partagée. Cette approche consiste à ce que chaque processus sélectionne l'événement local (ou celui reçu d'un processus qui l'influence) dont la date T est la date minimum de son échéancier. Le processus traite alors cet événement en planifiant un nouvel événement local ou à destination des processus qui l'influencent en actualisant sa date logique actuelle avec la date de l'événement traité. Lorsqu'un processus traite un événement de date T, il doit être sûr de ne pas recevoir d'autres événements de date $T' < T$, et donc respecter le principe de causalité. L'objectif de cette approche de synchronisation est donc de déterminer les événements « sûrs » à traiter. Les premiers alogrithmes utilisant l'approche conservative ont été proposés par Bryant et al. [42] et Chandy et al. [43].

Le principal avantage de l'approche conservative est qu'il n'est nécessaire de sauvegarder qu'un seul point de récupération par processus. Ce qui limite énormément la quantité de mémoire utilisée pour les mécanismes de synchronisation. En revanche, l'approche conservative possède une contrainte qui se caractérise par la non exploitation totale du parallélisme. En effet, si un événement $E1$ exécuté sur un processus $P1$ peut potentiellement affecter (directement ou indirectement) un événement $E2$ exécuté sur un processus $P2$, alors l'approche conservative doit forcément exécuter les événements $E1$ et $E2$ de manière séquentielle et dans cet ordre pour respecter la contrainte de causalité. Si aucune causalité n'existe entre les événements ils auraient pu être traités de manière concurrente.

En contraste avec l'approche précédente, la violation de la contrainte de causalité est possible avec l'approche optimiste. Cette approche consiste à ce que chaque processus traite les événements à sa connaissance. Cette connaissance locale et partielle des événements à traiter, peut impliquer l'omission de certains événements provenant d'autres processus et donc ne pas respecter la causalité [44]. Lorsqu'une contrainte de causalité n'est pas respectée, cas où la réception d'un événement possède une date antérieure à la date actuelle locale du processus. Alors un retour en arrière est effectué par l'intermédiaire de mécanismes appelés mécanismes de *Rollback* [45] permet de traiter l'événement à une date cohérente.

Le principal avantage de l'approche optimiste est qu'elle exploite au maximum le parallélisme, c'est à dire le traitement concurrent des événements. En revanche, cette approche optimiste implique de sauvegarder plusieurs points de récupération par processus pour garantir le bon déroulement des mécanismes de *Rollback*. Ainsi, la place mémoire utilisée pour stocker les points de récupération peut s'accroître rapidement. De plus, dans certains cas, cette approche n'est pas efficace si de nombreuses dépendances entre les processus sont présentes. En effet, dans ce cas, le temps d'exécution passé à effectuer des *Rollback* devient très important et impacte l'exécution.

Pour résumer, la performance offerte par ces approches et le choix d'une approche sont fortement liés au type d'application distribuée visée, suivant que l'application possède de nombreuses inter-dépendances ou non entre les processus.

Les modes *Ghost* et non *Ghost*

Dans le contexte parallèle, lors de la distribution d'un programme sur plusieurs processus, il est souvent nécessaire de copier des parties d'information sur chacun des processus afin de garantir une continuité dans la structure de donnée qui est distribuée. En effet, chaque processus participant à l'exécution reçoit une partie des informations du programme dont il est responsable. Cependant, il est souvent nécessaire d'utiliser des informations détenue par d'autres processus pour réaliser un calcul à cause des dépendances.

Dans la littérature, on distingue deux modes de fonctionnement, le mode *Ghost* et le mode non *Ghost*. Le mode *Ghost* consiste à utiliser les données calculées auparavant, au pas de temps ou à l'itération précédente, pour réaliser le calcul. Les résultats des calculs sont, eux, mémorisés dans une autre copie de travail. De ce fait, les processus sont obligés de conserver à la fois ces données calculées auparavant, le *Ghost*, et les résultats des calculs, la copie de travail. De cette manière, les informations du *Ghost* ne sont accessibles qu'en lecture et ne sont pas les dernières valeurs calculées. Ce modèle est une référence dans les simulations utilisant la méthode des éléments finis. En contraste avec le mode *Ghost*, le mode non *Ghost* consiste à ne disposer que d'une seule instance des données, dans laquelle les résultats sont directement reportés. De ce fait certains calculs sont réalisés directement avec la version des données calculée lors du pas de temps ou de l'itération courante. Avec ce mode, les informations sont accessibles en lecture et en écriture mais nécessitent des mécanismes de synchronisation pour les données détenues par d'autres processus.

Architecture parallèle hybride

Plus récemment, la popularisation des processeurs multi-coeurs et l'apparition de solutions d'exécution comme les cartes graphiques (*GPU*) ont favorisé l'apparition de parallélisations dites hybrides, mettant à œuvre dans un même programme plusieurs modèles de programmation différent comme OpenMP, MPI ou le GPGPU (General Purpose Computing on Graphics Processing Units). En effet, la carte GPU est elle aussi une architecture parallèle redoutable. Habituellement destinées au traitement d'image, elles sont de plus en plus utilisées pour le calcul intensif. Cela s'explique par le fait que l'architecture interne des GPU est différente des CPU et propose plusieurs centaines de cœurs contrairement à un CPU qui n'en propose qu'une vingtaine pour les meilleurs processeurs. L'utilisation de Cluster GPU nécessite une programmation particulière, qui repose sur des langages tels que CUDA ou OpenCL, pour s'adapter à l'architecture.

Plus récemment, une architecture hybride est apparue : les co-processeurs de type XeonPhi. Développée par Intel, cette architecture dite *many-core* est basée sur des processeurs Xeon. Elle permet d'obtenir de meilleures performances sans effectuer de grosses modifications pour réaliser le portage d'un programme CPU puisque le jeu d'instruction est le même que pour des processeurs classique. Le modèle de programmation est donc lui aussi adapté. Il permet l'utilisation hybride de MPI et de OpenMP simultanément.

Les auteurs Laville et al. proposent un framework pour développer des simulations multi-agents pour architectures *many-core* dans [46].

1.2.3 Évaluation des performances d'un programme parallèle

Lors de la parallélisation d'un programme, il existe plusieurs métriques qui permettent de quantifier l'efficacité ou le gain de ce programme lorsqu'il s'exécute en parallèle. Ces mesures permettent d'obtenir la qualité de la parallélisation ainsi que le comportement de l'application dans un contexte parallèle. Nous en détaillons quelques unes dans cette partie.

L'accélération ou Speed-up

On définit habituellement l'accélération comme étant une mesure du gain de performance (vitesse d'exécution) acquis par l'exécution parallèle d'un programme en comparaison à son exécution séquentielle. Pour calculer l'accélération d'un programme on considère un temps de référence puis pour obtenir l'accélération, il faut diviser le temps de référence par le temps mis par l'application parallèle pour s'exécuter sur p processus. Plus formellement on définit, l'accélération sur p processus par $A(p) = T(1)/T(p)$ où A est l'accélération, $T(1)$ est le temps d'exécution séquentiel (un processus) et $T(p)$ est le temps d'exécution sur p processus.

Plus généralement, l'accélération peut également être définie avec, comme temps de référence, le temps d'une exécution parallèle. En particulier, dans le cadre de la mesure de l'extensibilité d'un algorithme parallèle, les données d'entrée peuvent être de taille trop importantes pour la mémoire d'un seul, ou d'un petit nombre, de processus. Dans ce cas, le calcul de l'accélération s'effectue de la manière suivante : $A(p) = T(p_ref)/T(p)$ où A est l'accélération, $T(p_ref)$ est le temps d'exécution parallèle sur le nombre de processus de référence et $T(p)$ est le temps d'exécution sur p processus.

L'efficacité

On définit l'efficacité par $E(p) = A(p)/p$, où $A(p)$ est l'accélération de l'application sur p processus et p le nombre de processus utilisés. Ceci donne un pourcentage d'efficacité de la parallélisation sachant que 1 représente la parallélisation linéaire ou idéale.

La montée en charge

La montée en charge permet de mettre en avant le comportement d'une application parallèle face à l'augmentation de la charge de travail ou la quantité de calcul. Pour cela, un nombre fixe de processus p à utiliser est défini et l'application est exécutée plusieurs fois en augmentant à chaque fois la taille des données.

1.2.4 Les limites de la parallélisation

Plusieurs facteurs limitent les gains de performances d'une exécution parallèle : les dépendances entre les données, les ressources du programme exécuté et le temps de communication supérieur au temps de calcul à réaliser. Dans ces cas, le temps d'exécution d'une application parallèle peut considérablement augmenter et rendre la parallélisation inefficace et non avantageuse. En revanche, si l'application peut être découpée en tâches indépendantes, elle est dite *Embarrassingly parallel*, c'est à dire qu'elle se prête parfaitement à la parallélisation.

Un autre facteur limitant les performances est du à la distribution des calculs sur les différents processus. La charge de calcul que chaque processus effectue doit au maximum être répartie de façon équitable pour permettent de diminuer le temps total de l'exécution du programme. Des mécanismes d'équilibrage de charge dynamique permettent d'adapter la charge en redistribuant dynamiquement les calculs de manière équitable entre les processus.

Les systèmes multi-agents se prêtent bien à la parallélisation car chaque agent est autonome et exécute ses propres actions malgré quelques dépendances et communications.

La suite des travaux de ce manuscrit se concentre sur l'exécution de modèles SMA comportant un grand nombre d'agents réactifs dans un contexte parallèle de type cluster CPU ou supercalculateur (mémoire distribuée, paradigme SPMD, modèle de programmation MPI). Dans le chapitre suivant nous comparons les plateformes multi-agents parallèles existantes et nous détaillons les contraintes que ces plateformes imposent.

Chapitre 2

Les plateformes multi-agents et le parallélisme

Sommaire

La plupart des plateformes multi-agents ne prennent pas en compte nativement l'exécution de simulation multi-agents en parallèle. Les différentes approches possibles pour distribuer ou paralléliser une simulation multi-agents passent soit par le développement d'un modèle dédié [47], soit par l'implémentation d'une surcouche [48] qui va permettre d'ajouter des fonctionnalités à la plateforme telles que la distribution des agents, la synchronisation, la communication entre les agents, etc... Ces approches sont souvent très complexes car elles requièrent des connaissances en programmation. La plupart des modèles multi-agents sont développés par des non informaticiens ou des programmeurs non spécialistes. Bien que certains travaux [49, 50] proposent des approches pour faciliter la distribution des systèmes multi-agents, la mise au point d'une simulation parallèle, qu'elle soit agent ou non, requiert des compétences spécifiques pour s'exécuter efficacement sur une machine parallèle. Pour cette raison, il est donc plus facile d'utiliser des plateformes multi-agents parallèles et distribuées (PDMAS : Parallel and Distributed multi-agents Simulation). Une PDMAS fournit tous les outils et mécanismes nécessaires pour développer et exécuter facilement des modèles agents sur des ressources parallèles comme présenté dans le Chapitre 1 à la Section 1.2. Comparé aux grand nombre de plateformes multi-agents existantes, seulement quelques plateformes multi-agents parallèles et distribuées proposent nativement l'exécution de modèles en parallèle. Les fonctionnalités proposées nativement sont habituellement, la collaboration entre les exécutions de plusieurs nœuds physiques, la distribution des agents entre les nœuds, la communication entre les agents et la synchronisation entre plusieurs nœuds de calcul.

Durant l'analyse de la littérature, nous n'avons pas trouvé de comparatif sur les PDMAS à l'exception de l'article des auteurs Coakley and al. [51]. Dans [51], les auteurs effectuent un comparatif basé sur des critères qualitatifs tels que le langage d'implémentation mais les auteurs n'effectuent pas d'analyses approfondies sur les mécanismes internes des plateformes. Pour cette raison, nous proposons un comparatif basé sur des critères qualitatifs en explorant les mécanismes internes des plateformes et aussi un comparatif de performances dans un environnement d'exécution HPC.

2.1 Les plateformes multi-agents parallèles existantes

Après avoir effectué une recherche bibliographique en utilisant des mots clefs et en suivant les liens des différents articles sur le sujet, nous avons été capables d'établir une première liste de dix plateformes PDMAS ou projets de plateformes PDMAS. A notre connaissance, il n'y a pas d'autres plateformes **disponibles** et **fonctionnelles** qui proposent un support générique pour les simulations multi-agents parallèles.

Nous effectuons un bref descriptif des dix PDMAS concernés : RepastHPC [52], D-Mason [53], Pandora [54], FLAME [51], JADE [55], PDES-Mas [56, 57], SWAGES [58], Ecolab [59], MACE3J [60] et ABM++.

RepastHPC

RepastHPC [52] est développée par le laboratoire national d'Argonne (USA) (`http://repast.sourceforge.net/repast_hpc.html`). Elle fait partie de la série de plateformes de simulation multi-agents RepastJ, RepastSimphony. Elle est spécialement conçue pour les environnements haute performance. RepastHPC reprend le core de RepastSimphony, c'est-à-dire l'utilisation des projections (grid, network) et des contextes mais en les adaptant à l'environnement parallèle et distribué. Le langage utilisé pour implémenter une simulation agent est le $C++$ ou le ReLogo, un dérivé du langage Netlogo. Pour la communication, la plateforme RepastHPC utilise MPI par l'intermédiaire de la librairie Boost [2].

D-Mason

D-Mason (Distributed Mason) [53] est développée par l'université de Salerne (`http://isis.dia.unisa.it/projects/dmason/`). D-Mason est la version distribuée de la plateforme de simulation multi-agents Mason. Les auteurs ont souhaité élaborer la version distribuée de Mason afin de fournir une solution qui n'oblige pas les utilisateurs à réécrire les simulations qu'ils ont déjà développées pour repousser le nombre maximal d'agents. D-Mason utilise JMS ActiveMQ pour la communication, malgré le fait qu'elle ne soit pas la plus extensible des solutions dans un environnement HPC. D-Mason utilise le langage Java.

Pandora

Pandora [54] est développé par le groupe de recherche de simulation sociale du Centre Supercomputing Barcelone (`https://github.com/xrubio/pandora`). Elle a été explicitement programmée pour permettre l'exécution de simulations multi-agents à grande échelle et elle est capable, selon la littérature, de traiter des milliers d'agents avec des actions complexes. Pandora possède un support pour un système d'information géographique (SIG) pour faire face à des simulations dans lesquelles les coordonnées spatiales sont pertinentes. Pandora utilise le langage $C++$ pour définir et implémenter un modèle de simulation agent. En revanche pour la communication, Pandora génère automatiquement du code MPI.

Flame

FLAME [51] est développée principalement à l'Université de Sheffield (`http://www.flame.ac.uk`). FLAME fournit des spécifications sous la forme d'un cadre formel qui peut être utilisé par les développeurs pour créer des modèles et des outils. FLAME permet la parallélisation à l'aide de MPI. L'implémentation d'une simulation FLAME est basée sur la définition de X-machines [61] qui sont définies comme des automates d'états finis avec de la mémoire. Ils peuvent recevoir et envoyer des messages à l'entrée et à la sortie de chaque état.

Jade

JADE [55] est développée par le laboratoire Télécom Italia (`http://jade.tilab.com/`) et vise à simplifier la mise en œuvre de systèmes multi-agents distribués à travers un middleware qui se conforme aux spécifications de FIPA [55] et à travers un ensemble d'outils qui prennent en charge les phases de débogage et de déploiement. La plateforme peut être distribuée sur plusieurs ordinateurs et la configuration peut être contrôlée à partir d'une interface graphique à distance. La configuration peut même être changée au moment de l'exécution par des agents mobiles d'une machine à une autre en cas de besoin. L'implémentation d'un agent se fait par l'intermédiaire du langage Java, tandis que la communication s'effectue à l'aide de plusieurs supports comme Java-RMI, HTTP ou IIOP qui permettent l'utilisation du standard FIPA .

PDES-MAS

PDES-MAS [56, 57] est développée par le laboratoire Systèmes Distribués de l'Université Birmingham. PDES-MAS utilise le principe de simulation à événements discrets avec une approche optimiste de la synchronisation. Chaque agent est modélisé comme un Agent Logical Process (ALP). L'environnement, appelé état partagé, est maintenu par un ensemble de structures arborescentes dynamiques et transparentes reconfigurées à l'aide des ALP. La communication s'effectue grâce à des Communication Logical Process (CLP). PDES-Mas utilise MPI comme couche de communication.

SWAGES

SWAGES [58] est développée par l'Université de Notre Dame (USA). SWAGES permet la parallélisation et la distribution dynamique et automatique de simulations multi-agents dans un environnement hétérogène. Ce framework est généralement divisé en deux parties, la partie serveur et la partie client. SWAGES permet le développement de simulations agents avec l'un des langage suivant : Poplog, C ou encore Java. Les bibliothèques de communication utilisées sont XML-RPC et SSML.

Ecolab

Ecolab [59] est développé par l'Université of New South Wales. Ecolab est une plateforme de modélisation et de simulation à base d'agents pour les programmeurs C++. Ecolab est fortement influencée par la conception de la plateforme Swarm. Ecolab utilise MPI comme couche de communication.

MACE3J

MACE3J [60] est développée par l'université de l'Illinois. MACE3J est une plateforme multi-agents basée sur le langage Java. MACE3J peut être exécutée sur des stations de travail multi-processeurs aussi bien que dans des environnements plus grands de type cluster. La conception de

MACE3J est multi-grain, mais accorde une attention particulière à la simulation de très grandes quantité d'agents. MACE3J présente un important degré d'évolutivité. Aucune information n'a été trouvée sur la couche de communication utilisée.

ABM++

ABM++ est développée par le Los Alamos National Laboratory. Ce framework permet l'implémentation de modèles agents à l'aide du langage C++ et offre une interface permettant d'accéder à la couche de communication MPI. ABM++ propose les fonctionnalités nécessaires pour exécuter des modèles agents sur des architectures à mémoire distribuée. Ce framework offre la possibilité d'animer les modèles à l'aide d'événements ou à l'aide de pas de temps.

2.2 Analyse qualitative des plateformes existantes

Dans cette section, nous présentons une comparaison qualitative des différentes plateformes multi-agents parallèles et distribuées. En utilisant, la documentation fournie par chacune des plateformes, nous effectuons une comparaison de leurs propriétés ainsi que des fonctionnalités proposées pour implémenter les modèles. Il est important de noter que nous limitons notre analyse aux informations fournies dans les documentations disponibles. Cette analyse qualitative se concentre d'abord sur les propriétés générales qui sont utiles lors de l'élaboration d'un modèle agent, comme le langage de programmation ou la manière dont les agents sont synchronisés. Nous analysons également les fonctionnalités apportées par les plateformes pour animer les modèles agents parallèles.

2.2.1 Méthodologie

Dans cette section, nous présentons les Trois ensembles de critères définis pour comparer, analyser et évaluer chacune des plateformes.

Le premier ensemble de critères aborde le développement des agents ainsi que les propriétés des agents dans la plateforme. Ces critères permettent de renseigner la manière dont les agents sont implémentés et représentés. A l'aide de ces critères nous cherchons à caractériser le modèle de programmation proposé par la plateforme. Les critères que nous utilisons sont les suivants :

1. Langage de programmation : le langage utilisé pour développer les modèles. Il est important de noter que certaines plateformes proposent différents langages pour la définition du modèle (interaction, état des agents) et pour l'implémentation des comportements des agents.
2. Représentation des agents : le concept utilisé pour représenter un agent dans le langage de programmation. Chaque plateforme propose l'animation de modèle multi-agents, et de fait propose le paradigme agent. Comme la majorité des plateformes utilise des langages orientés objets, le paradigme agent est généralement basé sur le paradigme objet. Cependant, dans certaines plateformes les agents peuvent être représentés non seulement par un objet, mais aussi par un autre ensemble de données qui complètent la description de l'agent.
3. État d'un agent : ce critère donne la représentation de l'état de l'agent dans la plateforme habituellement constitué des données locales de l'agent.
4. Comportement des agents : il constitue la partie dynamique d'un agent. La représentation des comportements d'un agent peu prendre différentes formes, ce qui est important lorsque l'on implémente un modèle.

5. Identification des agents : les agents sont identifiés dans une simulation. Cette identification est nécessaire pour envoyer des messages ou pour interagir avec d'autre agents. Comme les agents seront distribués dans un environnement parallèle, il est important d'avoir une identification d'agent appropriée.

Le deuxième ensemble de critères concerne les propriétés générales des plateformes du point de vue de la simulation et des garanties fournies. Ces critères sont les suivants :

1. Synchronisation des agents : les simulations multi-agents sont entraînées par un ordonnanceur qui anime le modèle. Deux principales approches sont généralement considérées comme : le pas de temps (time-driven) et les événements (event-driven). Ces deux approches ont été précédemment présentées dans le Chapitre 1 à la section 1.1.3.
2. Événement simultané : dans les SMA séquentiels, les événements et comportements des agents sont exécutés les uns après les autres. Dans une plateforme parallèle comme les PDMAS, plusieurs exécutions sont effectuées simultanément.
3. Reproductibilité : La simulation d'un modèle doit être un processus déterministe et nous devrions toujours obtenir le même résultat avec un ensemble de données et de paramètres identiques en entrée. A noter toutefois que ce critère peut atteindre sa limite dans un contexte parallèle. Dans le cas de l'exécution de simulations sur des machines différentes avec des vitesses différentes, il est difficile de mettre en œuvre une reproductibilité absolue. L'ordre d'exécution des agents peut être différent quand ils ne s'exécutent pas sur le même nœud et que les nœuds ne sont pas synchronisés.
4. Nombre aléatoire : les comportements des agents reposent souvent sur des choix aléatoires. Un générateur de nombres aléatoires est généralement fourni avec les plateformes multi-agents. Les propriétés du générateur de nombres aléatoires sont donc importantes lors du choix d'une plateforme multi-agents.

Le troisième ensemble de critères regroupe les caractéristiques des plateformes concernant le parallélisme. Comme ce comparatif vise à évaluer les plateformes parallèles, ces critères sont très importants. Il existe plusieurs façons de mettre en œuvre un PDMAS et les choix de mise en œuvre influent sur la portée et la qualité de la plateforme. Tandis que les SMA centralisés sont basés sur une instance de la plateforme, les PDMAS sont composés de plusieurs instances, une par cœur en général, et doivent collaborer pour offrir un système global. Cette collaboration repose sur la synchronisation et la communication entre les instances. Les critères pris en compte sont les suivants :

1. Interaction agent : dans certains modèles les agents doivent interagir les uns avec les autres. Ces interactions peuvent être limitées au champ de perception de l'agent ou non. Dans un contexte parallèle, les interactions entre les agents sont plus difficiles à mettre en œuvre que dans un contexte centralisé car les agents ne peuvent pas toujours accéder directement à la mémoire des autres agents qui sont exécutés sur un autre processus. Ce critère exprime le support apporté par la plateforme en ce qui concerne les communications entre les agents.
2. Couche de communication : la plupart des plateformes repose sur des couches de communication déjà existantes. Les propriétés de la couche de communication ont un impact sur les propriétés de la plateforme et donc sur l'animation du modèle. Certaines couches de communication telles que MPI sont plus orientées cluster HPC alors que d'autres comme RMI sont moins efficaces et plus adaptées pour les architectures moins couplées telles que les réseaux de stations de travail.
3. Synchronisation : lors d'une animation de simulation multi-agents, l'ordonnanceur doit garantir que l'ordre de traitement des événements respecte l'ordre du temps simulé. Sur les machines parallèles chaque nœud a cependant sa propre horloge de sorte que des procédures

de synchronisation doivent être implémentées pour assurer l'ordre du temps et respecter les contraintes de causalité. Une synchronisation conservative, permet un meilleur équilibre de la charge et donc des gains de performance. Les approches de synchronisations conservative et optimiste ont été précédemment présentées dans le Chapitre 1 à la section 1.2.2.

4. Équilibrage de charge : pour obtenir de bonnes performances lors de l'animation d'une simulation, tous les processeurs alloués doivent être utilisés efficacement. Lorsque la charge des processeurs n'est pas équilibrée, l'efficacité est de fait impactée. L'équilibrage de charge peut être géré par l'application ou automatiquement par le PDMAS. Le critère d'équilibrage de charge fait référence à la prise en charge automatique de l'équilibrage de charge dans la plateforme.
5. Architecture : il y a plusieurs architectures possibles pour distribuer les processus sur les processeurs et gérer les échanges d'informations entre ces processus. Ce critère est défini comme *distribué* si tous les processus sont autonomes et participent à la simulation. Il est défini sur *maître/esclave* si un processus dirige les autres durant l'exécution de la simulation.
6. Extensibilité (Scalabilité) : ce critère fait référence à la possibilité d'exécuter de grandes simulations, en termes de nombre de processeurs ou nœuds utilisés. C'est un critère important car il mesure la capacité de la plateforme à passer à l'échelle.

Analyse

La Table 2.1 donne une représentation synthétique de la comparaison pour le développement d'un agent ainsi que de ces propriétés dans la plateforme. La plupart des plateformes utilisent des langages classiques comme C-C++ ou Java pour définir des agents, à l'exception de la plateforme Flame qui utilise le langage XMML. Le langage XMML est une extension du langage XML conçu pour définir des X-Machines. On notera que la plateforme RepastHPC, en plus du langage de programmation C++ propose le langage Logo qui est un langage agent largement répandu. Repast-Logo ou R-Logo est la mise en œuvre de Repast Logo pour C++. Il permet de simplifier la réalisation de la simulation au prix d'une plus faible puissance d'expression par rapport à C++.

Les agents sont généralement définis comme des objets dont les méthodes représentent les comportements et les variables représentent les états. Tous les agents sont identifiables dans la simulation à l'aide d'un identificateur unique. Un conteneur d'agents rassemble tous les agents. Ce conteneur est découpé et distribué dans le cas d'une exécution en parallèle. La mise en œuvre des agents est différente pour la plateforme Flame qui n'utilise pas le concept d'objet pour définir un agent mais utilise des automates appelés X-Machines [62]. Dans une X-Machine, un comportement est représenté par un état de l'automate et l'ordre d'exécution entre les comportements est représenté par des transitions. Cette différence modifie la logique de programmation d'un modèle, mais n'induit pas de limitation par rapport à d'autres plateformes, car les agents sont au final codés avec le langage C.

La Table 2.2 regroupe les critères sur les propriétés générales des plateformes en ce qui concerne la simulation et les garanties fournies. Alors que la plupart des plateformes utilisent le pas de temps pour animer la simulation et discrétiser le temps, la plateforme JADE cible plutôt des applications d'agents distribués et elle ne propose donc pas de synchronisation, ni même de notions de temps c'est la raison pour laquelle elle est marquée Non APplicable (NAP) pour ce critère dans le tableau. Des travaux récents proposent un cadre de contrôles pour les SMA à l'aide du temps pour la plateforme JADE [64], ce qui prouve que la plateforme peut être complétée pour gérer le temps.

Plateforme	Langage de programmation	Représentation des agents	État d'un agent	Comportements des agents	Identification des agents
RepastHPC	C++/RLogo	Objets	Variables	Méthodes	ID unique
D-Mason	Java	Objets	Variables	Méthodes	ID unique
Flame	XMML/C	CSXM/C [62, 51, 63]	Acyclic State Machine [51]	Transition / State / Functions [63, 51]	ID unique
Pandora	C/C++	Objets	Variables	Méthodes	ID unique
JADE	Java	Objets	Variables	Méthodes	ID unique
PDES-MAS	C/C++	Objets / Agent Logical Process (ALP)	Variables	Méthodes	ID unique
Ecolab GraphCode	C++	Objets	Variables	Méthodes	ID unique
SWAGES	Poplog/Java [58]	Objets	Variables	Méthodes	ID unique
MACE3J	Java	Objets	Variables	Méthodes	ID unique
ABM++	C/C++	Objets	Variables	Méthodes	ID unique

TABLE 2.1 – Critères concernant le développement et les propriétés des agents

Pour l'animation des agents, toutes les plateformes utilisent l'approche par pas de temps sauf RepastHPC et PDES-MAS qui sont basées sur l'animation par événement et JADE qui ne fournit pas de gestion du temps ni de synchronisation comme expliqué précédemment. RepastHPC permet cependant de fixer une périodicité à chaque événement prévu, de sorte qu'il est facilement possible de produire des simulations animées par pas de temps malgré un fonctionnement par événements.

Dans les plateformes parallèles comme les PDMAS plusieurs exécutions sont faites simultanément, il est important de noter que seule Flame ne supporte pas les exécutions d'événements simultanés.

La simulation d'un modèle doit autant que possible être déterministe, c'est-à-dire que différentes exécutions avec le même ensemble de paramètres doivent toujours produire les mêmes résultats. La plupart des plateformes, prennent en compte la reproductibilité. Les plateformes pour lesquelles nous ne trouvons pas d'information sur la propriété de reproductibilité sont marquées comme Non Disponible (ND) pour cette propriété.

Le générateur de nombres aléatoires le plus utilisé est Mersenne Twister. Pour différentes plateformes, nous n'avons pas trouvé de données sur le générateur de nombres aléatoires. Elles sont marquées comme Non Disponible (ND) pour cette propriété.

La Table 2.3 résume les critères des plateformes en ce qui concerne le parallélisme. Globalement, nous pouvons noter que toutes les plateformes étudiées répondent aux exigences liées aux développement de simulations parallèles.

Toutes les plateformes permettent aux agents de communiquer en local, c'est à dire avec avec des agents sur le même processus, et de manière distante, avec copie des agents sur différents processus. Les plateformes D-Mason et Pandora proposent l'invocation de méthodes à distance pour communiquer avec d'autres agents, tandis que les autres plateformes utilisent des messages pour communiquer entre agents.

Les couches de communications de la plupart des plateformes reposent sur MPI. Cela n'a rien de surprenant pour des plateformes qui ciblent les systèmes parallèles et HPC. MPI est

Plateforme	Synchronisation des agents	Événement simultané	Reproductibilité	Nombre aléatoire
RepastHPC	Event-driven	Oui [51]	Yes	Mersenne Twister [65]
D-Mason	Time-driven	Oui [51, 53]	Oui	Mersenne Twister
Flame	Time-driven	Non [51]	ND	Marsaglia [66]
Pandora	Time-driven	Oui	Oui [54]	ND
JADE	NAP	NAP	ND	ND
PDES-MAS	Event-driven	Oui	Oui	ND
Ecolab GraphCode	Time-driven	Oui [51, 59]	Non	UNU.RAN [67]
SWAGES	Time-driven	Oui	ND	Mersenne Twister
MACE3J	Time-driven	Oui	Oui [60]	Own random generator
ABM++	Time-driven	Oui	ND	ND

TABLE 2.2 – Critères concernant la simulation et les garanties fournies par les plateformes

principalement utilisé sur ce type d'architecture comme nous l'avons présenté dans le Chapitre 1 à la section 1.2. Il est important de noter que la plateforme D-Mason est basée sur le service de communication JMS bien qu'il ne soit pas la solution la plus évolutive pour les environnements parallèles et distribués. Une version MPI de D-Mason est en cours de développement. Enfin, le support de communication de la plateforme Jade est basé sur Java-RMI, HTTP ou IIOP qui ne sont pas adapté aux applications parallèles car ils sont basés sur des appels synchrones et ne permettent pas une utilisation efficace des réseaux de haute performance (par exemple InfiniBand).

Pour exploiter efficacement la puissance des ressources parallèles, la charge de calcul doit être équilibrée entre les différents noeuds. Il existe différentes façons d'équilibrer la charge de calcul. Elle peut etre équilibrée au début de la simulation (équilibrage statique) ou alors adaptée lors de l'exécution (équilibrage dynamique). Un équilibrage de charge dynamique est généralement préférable car il offre une meilleure adaptation en cas de variation de la charge lors de l'execution du modèle, mais il peut aussi etre soumis à l'instabilité. La plupart des plateformes utilisent l'équilibrage de charge dynamique, sauf les plateformes Jade et Flame. Dans [68] les auteurs proposent un moyen d'utiliser l'équilibrage de charge dynamique avec la plateforme Flame.

Pour finir, plusieurs architectures peuvent être utilisées pour distribuer les processus sur les processeurs. L'architecture la plus utilisée est distribuée, c'est à dire que tous les processus sont autonomes. Seul D-Mason utilise une architecture Maitre/Esclave. Ce choix peut être expliqué par l'utilisation de la bibliothèque de communication ActiveMQ JMS. A noter que dans la dernière version de D-Mason, le schéma de communication est basée sur MPI et dans ce cas l'architecture est distribuée.

Enfin, notons que nous n'avons pas trouvé d'information sur la propriété d'extensibilité pour plusieurs plateformes. Pour cela nous marquons Non Disponible (ND) pour cette propriété.

Pour chaque plateforme, nous avons essayé de télécharger le code source ou exécutable et nous avons essayé de le compiler et de le tester avec les exemples et modèles fournis. Certaines plateformes n'ont pas été inclues dans notre étude parce qu'elles ne proposent pas de code source disponible ou téléchargeable (MACE3J [60], JAMES [69], SWAGES [58]) ou ne proposent qu'une version de démonstration (PDES-MAS [56, 57]), ou parce qu'il y a un réel manque de documentation (Ecolab [59]). Malheureusement, il n'a pas été possible de développer un modèle pour ces plateformes, et donc d'évaluer leur caractéristiques et performances dans un

Platform	Interaction agent	Couche de communication	Synchronisation	Équilibrage de charge	Architecture	Extensibilité
RepastHPC	Locale/Distante (Copy of others) [2]	MPI/Boost lib	Conservative	Dynamique	Distribué	1028 proc.
D-Mason	Locale/Distante (AOI) [53]	JMS ActiveMQ / MPI	Conservative	Dynamique	Master/Slave [53]	36 proc.
Flame	Locale/Distante (Agent-agent+Sphere d'influence) [51]	MPI	Conservative	Statique [51]	Distribué	432 proc. [63, 51]
Pandora	Locale/Distante [54]	MPI	Conservative	Dynamique	Distribué [54]	ND
JADE	Locale/Distante	RMI / IIOP / HHTP / JICP	ND	ND	Distribué [55]	NA
PDES-MAS	Locale/Distante (Communication Logical Process (CLP) et sphère d'influence)	MPI	Optimiste	Dynamique	Distribué	ND
Ecolab Graph-Code	Locale/Distante (Copie d'autres agents)	MPI	Conservative	Dynamique	Distribué	ND
SWAGES	Locale/Distante	SSML	ND	ND	Distribué	50 proc.
MACE3J	Locale/Distante (MACE3J Messaging System,MMS)	Objets / Interfaces	Conservative	ND	Distribué	48 proc.
ABM++	Locale/Distante (Copie d'autres agents)	MPI	Conservative	ND	Distribué	ND

TABLE 2.3 – Critères concernant le parallélisme des plateformes

environnement parallèle. Pour ces raisons, ces plateformes ne sont pas considérées dans la suite de l'analyse

Après cette première évaluation, on peut noter que certaines plateformes ont plutôt été conçues pour animer des simulations multi-agents, à savoir animer un modèle composés de nombreuses entités (agents) qui exécutent un comportement individuel, tandis que d'autres plateformes sont plus axées sur l'aide à la distribution pour des applications à base d'agents, à savoir, des applications qui permettent par exemple de fournir un service de surveillance d'équipements. Par conséquent, lors de l'implémentation des modèles, nous avons constaté que la plateforme Jade semble être davantage axée sur la surveillance d'équipements plutôt que sur l'animation de simulations agents à large échelle. Le site officiel de Jade présente la plateforme comme une base pour les *"peer-to-peer agent based applications"*. En effet, la plateforme n'est pas orientée HPC [70] car elle ne peut pas être exécutée en mode batch. Il est important de noter que l'implémentation de simulation multi-agents de taille moyenne est possible dans JADE comme indiqué dans [71]. Par conséquent la plateforme JADE n'est pas inclue dans la suite de la comparaison et nous nous concentrons uniquement sur les plateformes D-Mason, Flame, Pandora et Repast-HPC.

2.2.2 Analyse de la distribution

Le développement d'une simulation multi-agents parallèle n'est pas une tâche aisée même si cela est facilité par les supports fournis par la plateforme. Notre objectif ici est d'évaluer le support proposé par les plateformes pour les simulations de grande taille. Pour cela, nous nous concentrons sur les caractéristiques de distribution des plateformes pour définir des critères d'évaluation. Pour mieux comprendre la façon dont ces caractéristiques peuvent avoir un impact sur le développement du modèle et pour valider le fait que les plateformes sont opérationnelles, nous évaluons également le développement d'un modèle sur les quatre plateformes restantes.

Méthode

Les critères d'évaluation se concentrent sur les propriétés de distribution des plateformes. Lorsque que l'on implémente un modèle agent qui va être distribué, nous sommes confrontés à quatre questions principales :

1. La distribution : ce critère fait référence à la façon dont les agents sont distribués sur les processeurs ou sur les nœuds. Comme l'exécution de la simulation se fait en parallèle l'ensemble des agents doit être divisé entre les processeurs afin de bénéficier de toute la puissance de traitement disponible. La distribution des agents se fait par l'intermédiaire de la plateforme, mais cette distribution influence grandement les performances, il est donc important de savoir comment cette distribution est effectuée.
2. Communication : ce critère fait référence à l'implémentation de la couche de communication dans la plateforme. Comme pour toutes les applications parallèles un système de communication efficace est nécessaire pour obtenir des performances et des simulations extensibles.
3. Cohérence/synchronisation : lorsque les agents sont exécutés en parallèle, il faut gérer les accès concurrents aux données communes. La cohérence des données ou la synchronisation des accès et leurs propriétés sont une connaissance nécessaire lors de l'animation d'un modèle car elles définissent les limites du support proposé par la plateforme. Si aucune synchronisation n'est fournie, les accès simultanées doivent être évités dans le modèle, ou gérés par le développeur du modèle.
4. Équilibrage de charge : ce critère fait référence à l'équilibrage de charge fourni par la plateforme. Ce critère représente la partie dynamique des plateformes, c'est à dire la distribution ou la re-distribution des agents lors de l'animation de la simulation. Cette distribution ou re-distribution implique que la plateforme doit être capable de déplacer des agents d'un nœud à un autre au cours de l'exécution de la simulation. Ce déplacement est généralement appelé migration d'agent. La migration des agents induit que la plateforme doit être en mesure de rediriger les communications entre les processus. C'est la raison pour laquelle l'équilibrage de charge est un critère important à considérer.

Ces trois propriétés sont importantes pour comprendre comment modéliser et implémenter efficacement un modèle agent dans une plateforme parallèle agents mais aussi pour comprendre quelles sont les contraintes imposées à la conception du modèle. En outre, ces critères sont une façon de montrer l'étendue de chacune des plateformes en ce qui concerne leur soutien à la distribution. Malheureusement, pour comprendre réellement les supports et mécanismes proposés par les plateformes, une simple lecture de leur documentation n'est pas suffisante. Pour cette raison, nous avons mis en place un même modèle de référence défini au Chapitre 1 et à la section 1.1.5 que nous avons implémenté sur chacune des quatre plateformes. L'implémentation de ce modèle, nous a permis de comprendre le fonctionnement de ces plateformes au regard des propriétés. Nous avons également été en mesure de vérifier si toutes les fonctionnalités énoncées

dans la documentation étaient vraies. Comme nous avons pu mettre en œuvre notre modèle pour les quatre plateformes, nous pouvons alors considérer qu'elles offrent vraiment un environnement fonctionnel pour prendre en charge les simulations multi-agents parallèles.

Dans les sections suivantes, nous présentons une synthèse pour chaque plateforme au regard des quatre critères énoncés précédemment.

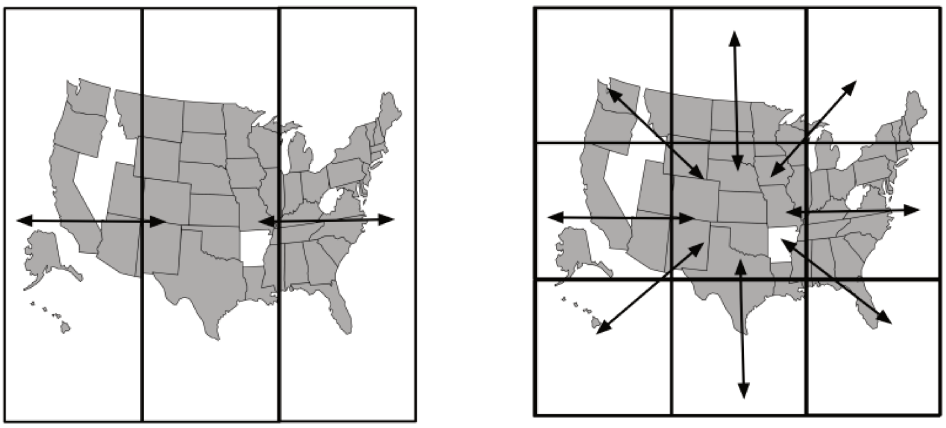
2.2.3 Distribution

La distribution est la base d'un modèle parallèle. La façon dont la distribution est mise en œuvre influe directement sur la complexité de l'élaboration du modèle mais aussi sur les performances d'exécution.

D-Mason

Du point de vue de la distribution, la plateforme D-Mason est une plateforme agents générique. Les agents peuvent ou non avoir une position dans un espace cartésien qui représente l'environnement. L'environnement est divisé en cellules ou partitions qui sont assignées à des processus lors de la distribution la simulation. D-Mason propose trois façons de distribuer une simulation sur plusieurs processus : les deux premières sont basées sur le partitionnement à base de grille ou cartésien et la troisième est basée sur le partitionnement de graphes. Des zones de recouvrement (Area of Interest or AOI) sont définies pour garantir la continuité de la perception des agents à travers les frontières des partitions.

Partitionnement de *Grid field* : La Figure 2.1 représente les deux mécanismes de distribution disponibles dans la plateforme D-Mason pour la distribution cartésienne. La distribution sur l'axe des Y consiste à diviser l'environnement en cellules verticales sur l'axe des Y (Figure 2.1(a)). La distribution XY consiste à diviser l'environnement non pas en cellules horizontales mais en carré utilisant les axes X et Y (Figure 2.1(b)).



(a) Dsitribution sur Y (b) Distribution sur XY

FIGURE 2.1 – Schéma de distribution d'une grille avec la plateforme D-Mason

La grille est distribuée en la découpant en cellules. Chaque cellule est ensuite assignée à des processus participant à l'exécution. Pour proposer un environnement continu, D-Mason utilise les zones de recouvrement, aussi appelées "Area of Interest" (AOI) comme présenté dans la Figure 2.2. Ce mécanisme consiste à copier une partie de chaque cellule sur les cellules adjacentes. La zone de recouvrement permet aux agents de garder un champ de perception complet même

quand l'environnement est divisé entre plusieurs processus. Pour maintenir cette continuité, des échanges d'informations avec les cellules voisines sont effectuées à chaque pas de temps.

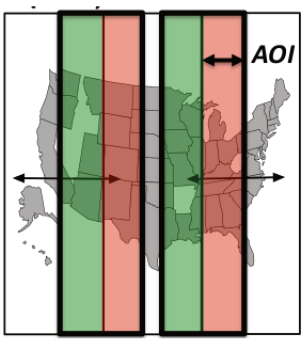


FIGURE 2.2 – Exemple de zones de recouvrement pour la distribution sur l'axe Y

Lors de la phase de distribution, seul l'environnement est considéré pour diviser la simulation. Les agents ne sont pas pris en compte dans cette phase. Cependant, la densité d'agents est prise en compte pour la phase d'équilibrage de charge dynamique.

Partitionnement de *Network field* : Le *Network field* propose l'utilisation de graphe pour représenter une structure entre les agents qui composent la simulation. Pour distribuer le graphes parmi plusieurs processus, les bibliothèques comme ParMetis [72] ou Metis sont utilisées. ParMetis est une bibliothèque basée sur le standard MPI qui implémente de nombreux algorithmes pour le partitionnement de graphes. ParMetis étend les fonctionnalités de Metis pour les graphes à large échelle. Nous n'avons malheureusement pas trouvé d'autres informations sur les mécanismes de distribution des *Network fields* dans la plateforme D-Mason car cette fonctionnalité a été récemment ajoutée.

Il est important de noter que dans la plateforme D-Mason, il est possible d'utiliser plusieurs type de structures dans une même simulation. En d'autre termes, il est possible d'utiliser une distribution cartésienne pour l'environnement tandis qu'un graphe peut être utilisé pour représenter les interactions entre agents. En revanche, dans ce cas la distribution sera uniquement basé sur le partitionnement cartésien.

Pour distribuer les simulations agents, la plateforme RepastHPC utilise des mécanismes appelés "Projection". Ces mécanismes sont adaptés de la plateforme agent Repast S. Les projections représentent l'environnement dans lequel les agents peuvent évoluer. Ces projections peuvent être de deux types : grille (*Grid*) ou graphe (*Network*). Elles permettent d'imposer une structure aux agents dans laquelle ils évoluent. De la même manière que la plateforme D-Mason, la plateforme RepastHPC définit des mécanismes de zones de recouvrement pour garantir la continuité du champ de perception des agents entre les processus.

RepastHPC

La distribution dans la plateforme RepastHPC présente de nombreuses similarités avec les propositions de la plateforme D-Mason. En effet, les projections de grilles et de graphes sont équivalentes aux partitionnements proposés dans la plateforme de D-Mason.

Partitionnement de *Projection Grid* : L'environnement est représenté par un espace cartésien. Pour distribuer le modèle sur plusieurs processus, l'environnement est divisé en cellules de taille égale. Ces cellules sont ensuite réparties sur les processus. Chaque processus est alors

responsable d'une sous-partie de la grille. Les sous-parties de la grille utilisent des zones de recouvrement pour garantir la continuité.

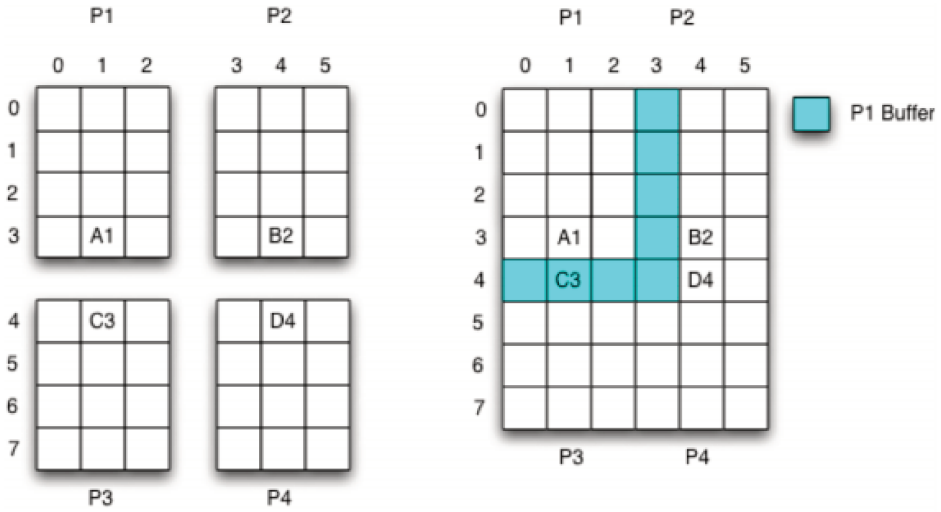


FIGURE 2.3 – Schéma de distribution d'une *Projection Grid* utilisant 4 processus dans la plateforme RepastHPC [2]

La Figure 2.3 représente un schéma de distribution d'une grille sur quatre processus. Les coordonnées de la grille sont de (0,0) à (5,7). Le processus $P1$ est responsable de la sous-partie (0.0) x (2.3), le processus $P2$ est responsable de la portion (3.0) x (5.3) et ainsi de suite. Dans cet exemple, la taille de la zone de recouvrement est de 1. Dans ce cas, le processus $P1$ contient donc une copie de la colonne 3 du processus $P2$ ainsi que la ligne 4 du processus $P3$.

Partitionnement de *Network Projection* **:** Les Network Projection [52] sont une manière de représentées une structure de graphe entre les agents qui composent la simulation. Afin de diviser le graphe sur plusieurs processus tout en maintenant les liens entre les sommets distribués sur différents processus, une copie des voisins est effectuée. Malheureusement, aucune information n'est disponible dans la documentation de la plateforme RepastHPC sur la manière dont la distribution du graphe sur plusieurs processus est effectuée, ce qui rend son utilisation difficile. La Figure 2.4 représente un schéma de distribution de graphe entre deux agents distribués sur 2 processus.

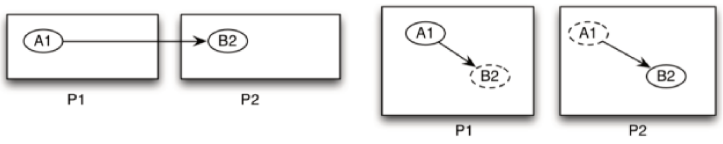


FIGURE 2.4 – Schéma de distribution du *Network Projection* utilisant 2 processus dans la plateforme RepastHPC [2]

Comme pour la plateforme D-Mason, il est possible d'utiliser plusieurs projections dans le même modèle : par exemple une projection de grille pour représenter un environnement géographique et un graphe pour représenter les interactions entre agents.

Flame

La distribution des agents dans la plateformes Flame diffère des deux plateformes précédemment présentés car la distribution est effectuée de manière statique [5]. La distribution est définie ou calculer au début de la simulation et ne change pas au cours de l'exécution. Les choix de distribution sont basés sur un graphe de dépendances de communication entre agents et a pour but de réduire les communications entre les processus. Deux manières de distribuer la simulation sont proposées : la manière cartésienne (*Geometric*) ou la manière Round Robin. Le choix de la distribution est défini par un paramètre au début de la simulation.

Partitionnement *Geometric* **:** Ce partitionnement consiste à distribuer les agents en fonction de leur position dans l'espace cartésien. Ce partitionnement est basé sur les coordonnées et peut être utilisé dans un environnement en deux dimensions ou trois dimensions. A noter que dans ce cas, l'environnement n'a pas été divisé sur la base d'une grille, l'environnement est toujours considéré comme continu et le partitionnement est seulement entraîné par la distribution. Le but de cette distribution est de regrouper les agents les plus proches car ils peuvent potentiellement communiquer ensemble et donc générer un grand nombre de communications.

Partitionnement *Round Robin* **:** Si les agents ne sont pas situés dans un espace cartésien, la plateforme Flame propose un mécanisme de distribution appelé Round Robin : les agents sont alors assignés à tour de rôle à des processus. Cette méthode de partitionnement ne prend pas en compte le comportement des agents mais il est possible d'ajouter un discriminant tel que le type de l'agent, qui va permettre d'influer sur la distribution. Par exemple, les agents de même type seront alors regroupés sur un ou plusieurs processus, si ces agents communiquent davantage entre eux qu'avec les agents d'autres types.

Pandora

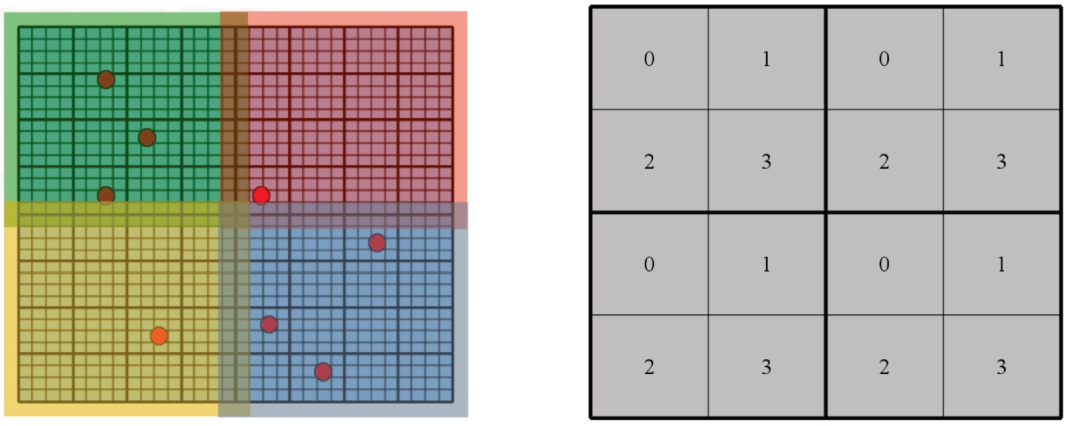
Dans la plateforme Pandora la distribution est effectuée par la division de l'environnement en plusieurs portions ou cellules comme pour les plateformes D-Mason ou RepastHPC. La Figure 2.5(a) provient d'une présentation de la plateforme Pandora. Cette figure présente la manière dont l'environnement est divisé et comment les zones de recouvrement sont utilisées afin de garder la continuité partielle de l'environnement même lorsque que les cellules sont distribuées sur plusieurs processus.

Il est important de remarquer qu'à l'exception de la plateforme Flame, toutes les autres plateformes utilisent une environnement cartésien pour distribuer dynamiquement les agents. Par conséquent, les simulations ainsi que les résultats seront donc liés à cette contrainte. Pour finir, seule la plateforme Flame propose uniquement une distribution statique des agents à partir de la configuration initiale et n'équilibre pas la charge durant l'exécution.

2.2.4 Analyse de la communication

Communication

La communication est l'une des fonctionnalités clef des plateformes d'exécution de modèles parallèles. Cette fonctionnalité repose principalement sur des appels de méthodes distantes ou sur l'échange de messages asynchrones.



(a) Schema of distribution (b) Execution schema

FIGURE 2.5 – Distribution et ordre d'exécution dans la plateforme Pandora [3, 4]

D-Mason

La communication entre les agents par l'intermédiaire de messages n'est pas implémentée dans la plateforme D-Mason. Les communications entre agents sont supportées par des appels de méthodes sur les agents cibles. Cette fonctionnalité de communication est cependant uniquement disponible pour les agents qui sont exécutés sur le même processus ou alors exécutés dans la zone de recouvrement. Si l'agent cible n'est pas exécuté sur le même processus ou n'est pas dans la zone de recouvrement alors il n'est pas possible de communiquer avec cet agent en utilisant le partitionnement par grille (*Grid field*). Cependant, en utilisant le partitionnement '*Network field*', il est possible de s'abstraire de cette limite. Ainsi la structure graphe permet à n'importe quel agent de communiquer avec un autre agent si il existe un lien entre eux.

Pour finir, la plateforme D-Mason propose une manière d'utiliser des paramètres globaux, c'est à dire des paramètres partagés par plusieurs processus. Cette fonctionnalité peut être utile pour diffuser une information à tous les agents de la simulation.

RepastHPC

La communication entre les agents par l'intermédiaire de messages n'est pas implémentée dans la plateforme RepastHPC. Les communications entre agents sont supportées par des appels de méthodes sur les agents cibles. Cette fonctionnalité de communication est cependant uniquement disponible entre les agents qui sont exécutés sur le même processus. Si l'agent cible n'est pas exécuté sur le même processus, l'agent émetteur doit explicitement faire une demande de copie de l'agent cible sur son processus. De cette manière, il est alors possible de faire un appel de méthode sur l'agent copie. Un point très important est que si un agent copie, est modifié, cette modification ne sera pas reportée sur l'agent original s'exécutant sur un autre processus.

Flame

Dans la plateforme Flame les agents ne peuvent pas envoyer directement de messages aux autres agents. Les communications sont implémentées par l'intermédiaire de tableau de messages ou *message boards*. Chaque processus possède sa propre table de messages qui est divisée entre les agents dont il est responsable. Les agents peuvent lire (recevoir) ou écrire (envoyer) des messages dans la partie du tableau de messages qui leur est attribué. Les agents peuvent envoyer et recevoir des messages de différents types qui sont définis dans le fichier XMML de la simulation. Il est

important de noter qu'un agent ne peut pas envoyer et recevoir un message dans le même comportement. Les communications doivent être implémentées entre deux comportements. La parallélisation de la distribution est donc basée sur la distribution des tables de messages comme cela est montré dans la Figure 2.6.

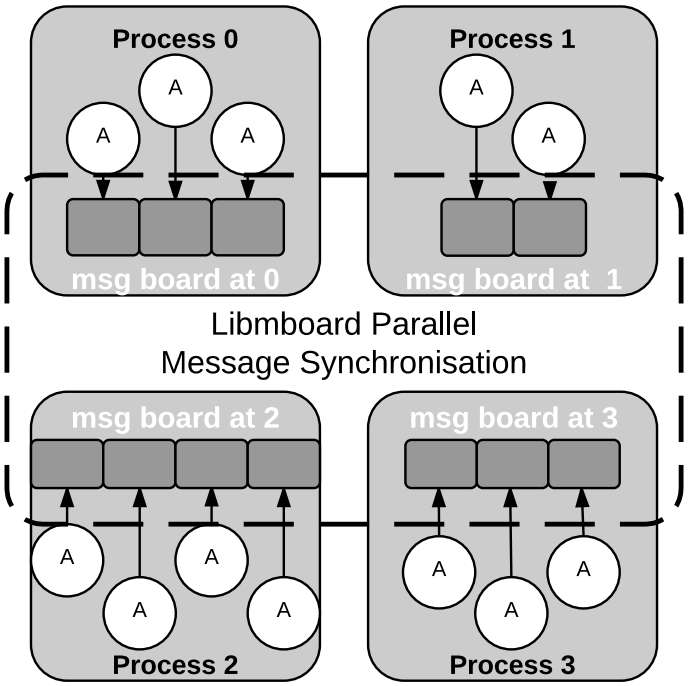


FIGURE 2.6 – Schéma de distribution des tableaux de messages dans la plateforme Flame [5]

Comme les agents peuvent être exécutés sur différents processus (en fonction de la distribution) et ne sachant où s'exécutent les agents, la plateforme Flame utilise la diffusion (broadcast) de messages pour atteindre l'agent ciblé par la communication. Il est cependant possible de limiter le nombre de destinataires par messages à l'aide de filtres. Dans ce cas, le message est uniquement diffusé (broadcast) à un groupe d'agents.

Les tableaux de messages sont gérés par un bibliothèque appelée *LibmBoard*. Cette bibliothèque est nécessaire au bon fonctionnement de la distribution, la création, la suppression, la synchronisation et à l'accès aux messages. La bibliothèque *LibmBoard* est basée sur le standard de communication MPI pour communiquer entre les processus et utilise les threads POSIX pour gérer les données et les communications inter-processus.

Pandora

Les communications dans la plateforme Pandora sont basées sur le standard de communication MPI et sur la bibliothèque μsik [73]. La bibliothèque μsik est conçue pour gérer les communications entre les processus et tout spécialement pour exécuter des simulations à événements discrets sur des ressources parallèles. La plateforme Pandora génère automatiquement le code pour éviter à l'utilisateur d'écrire le code MPI. Cependant, la plateforme Pandora limite la communication entre agents à leur champ de perception [3].

2.2.5 Analyse de la cohérence et synchronisation

La gestion de la cohérence et de la synchronisation diffère réellement entre les quatre plateformes : chaque plateforme résout le problème de cohérence à sa manière en proposant le plus de transparence possible pour le développeur.

D-Mason

La plateforme D-Mason implémente des mécanismes de synchronisation conservatif qui garantissent qu'il n'y a aucune erreur de causalité. Pour réaliser une synchronisation conservative, chaque pas de temps de la simulation est divisé en deux étapes : (1) la communication et synchronisation et (2) l'exécution de la simulation. Un barrière de synchronisation est présente à chaque fin de pas de temps. Les agents d'une cellule c ne peuvent pas exécuter le pas de temps i tant que les cellules voisines n'ont pas terminé d'exécuter la simulation du pas de temps $i-1$. A la fin d'un pas de temps, chaque cellule envoie aux cellules voisines les informations concernant les agents qui se situent dans la zone de recouvrements ou les agents qui doivent être migrés d'un processus à un autre. Pour le pas de temps i les comportements de la cellule c sont ainsi calculés à partir des les informations du pas de temps $i-1$ des cellules voisines.

La plateforme D-Mason propose différentes stratégies de communication pour la synchronisation. Les stratégies sont basée sur ActiveMQ et MPI [74]. La première version de D-Mason utilise Active MQ JMS avec le paradigme "Publisher/Subscriber". Dans la dernière version de D-Mason les auteurs ont implémenté ce même paradigme à l'aide de MPI. Dans cette version les processus MPI sont assignés à des *D-Mason workers* et des communications collectives sont utilisées pour communiquer entre les cellules de la simulation.

Le modèle de synchronisation utilise et respecte cinq assertions :

— chaque cellule crée son propre topic pour publier les mises à jour,

— chaque cellule souscrit à au moins un topic (de ses cellules voisines). Une cellule ne peut pas démarrer un nouveau pas de temps tant qu'elle n'a pas reçu les informations de mise à jour des topics auxquelles elle a souscrit,

— la souscription est statique : chaque cellule souscrit avant le lancement de la simulation,

— La publication sur un topic où plusieurs cellules ont souscrit, ne peut s'effectuer que si toute les mises à jour de chaque cellule sont disponibles,

— La publication est appelée uniquement à la fin du pas de temps.

Avec les cinq assertions définies ci-dessus, il y a trois types de stratégies de communication en fonction des routines de communications MPI utilisées : Bcast qui utilise la diffusion, Gather qui utilise la collecte de messages et Parallel qui utilise les routines envoi et réception pour assurer des communications en parallèle.

RepastHPC

La synchronisation entre les processus dans la plateforme RepastHPC est effectué uniquement dans quatre cas comme le présente les auteurs Collier et al. dans [2]. Premièrement, lorsqu'un processus a besoin d'une ou plusieurs copies d'agents présents sur un autre processus. Cette synchronisation est nécessaire pour maintenir la simulation dans un état cohérent. Deuxièmement, quand un processus possède un agent copie ou une arrête copie (dans le cas d'un graphe) d'un autre processus, les copies doivent être mises à jour avec les dernières informations de l'agent original. En troisième lieu, les zones de recouvrement d'une grille doivent être mises à jour à chaque fin de pas de temps. Et pour finir, quand un agent doit être entièrement migré d'un processus à un autre. Par exemple, lorsque l'agent se déplace dans l'environnement.

La plupart des mécanismes de synchronisation ne sont pas à la charge du programmeur. Les programmeurs ont seulement à développer un ensemble de méthodes qui définissent les informations nécessaires qui doivent être synchronisées dans les agents. Il y a deux méthodes à définir qui sont *Provider*, qui est utilisée pour sérialiser les informations avant de les envoyer et *Receiver*, qui est utilisée pour dé-serialiser les informations à la réception. Ces deux méthodes sont un moyen de sérialiser les informations d'un agent afin de synchroniser les données nécessaires lors de la migration.

Flame

La synchronisation entre les processus de la plateforme Flame est conservatrice et elle est réalisée à travers les tableaux de messages. Tous les échanges entre les agents étant effectués en utilisant des messages, la synchronisation repose donc sur la synchronisation des tableaux de messages. La synchronisation des tableaux de messages est effectuée en deux étapes : la demande de synchronisation puis l'exécution de la synchronisation. Dans la première étape, lorsqu'un processeur a terminé d'exécuter ses agents, il verrouille son tableau de messages et envoie aux autres processus une demande de synchronisation dans une file d'attente. Après cette étape, il est encore possible de faire des actions qui ne nécessitent pas l'utilisation du tableau de messages. Lorsque tous les processeurs ont verrouillé leurs tableaux de messages, l'étape suivante est effectuée. Cette étape d'exécution de la synchronisation s'effectue par échange de messages entre les tableaux. L'étape d'exécution de la synchronisation est une phase de blocage. Après ces deux étapes, les tableaux de messages sont débloqués et la simulation se poursuit.

Pandora

La synchronisation dans la plateforme Pandora est aussi conservatrive [54]. Comme dans les plateformes RepastHPC et D-Mason, les données et les agents situés dans les zones de recouvrement sont copiés et envoyés aux cellules voisines à chaque pas de temps afin de conserver des données à jour. La taille de la zone de recouvrement est définie comme étant la taille maximale du champ de perception d'un agent [75, 4]. Pour résoudre le problème de synchronisation des zones de recouvrement entre les processus - résoudre les conflits entre les agents sur les différents processus - la plateforme Pandora utilise une méthode différente de celle des autres plateformes. Chaque partie de l'environnement qui est distribué sur plusieurs processus est également divisée en quatre sous-parties égales, numérotées de 0 à 3, comme présenté dans la Figure 2.5(b). Au cours de la phase d'exécution d'agents tous les processus exécutent séquentiellement chacune des sous-parties. En d'autres termes, tous les processus exécutent la sous-partie 0, puis, la sous-partie 1 et ainsi de suite. De cette façon, il n'y a aucune possibilité de conflits car toutes les sous-parties ne sont pas adjacentes. Une fois qu'une partie est terminée, les modifications des zones de recouvrement sont envoyées aux cellules voisines. Lorsque toutes les sous-parties sont exécutées l'état de la simulation est sérialisée et un nouveau pas de temps peut être exécuté.

2.2.6 Analyse de l'équilibrage de charge

Nous présentons dans cette section les informations que nous avons trouvées à propos de la prise en charge et du fonctionnement de l'équilibrage de charge dans les plateformes testées.

D-Mason

L'équilibrage de charge dans la plateforme D-Mason est effectuée dynamiquement [76]. La méthode proposée peut être utilisée dans des environnements 2D et 3D. L'équilibrage de charge est mis en œuvre comme une étape supplémentaire à la fin d'un pas de temps, en plus de la

synchronisation et de la simulation. A la fin de chaque pas de temps, la densité en terme d'agents est calculée pour chaque cellule de l'environnement. Les cellules ayant la plus grande densité sont alors re-distribuées sur les cellules voisines afin de répartir la charge.

Flame

Il n'y a pas d'équilibrage dynamique de la charge dans la plateforme Flame. L'équilibrage de la charge est effectué de manière statique au début de la simulation à l'aide des distributions proposées. Il est important de noter que les auteurs Marquez et al. in [68] propose un schéma d'équilibrage de charge dynamique pour la plateforme Flame. Malheureusement il n'est pas encore implémenté dans la version principale de Flame.

RepastHPC

Nous n'avons pas trouvé d'informations concernant les mécanismes mis en œuvre dans les plateformes RepastHPC pour effectuer l'équilibrage dynamique de charge. C'est la raison pour laquelle nous ne commentons pas cette propriété.

Pandora

Bien que la plateforme Pandora soit utilisée dans différents projets comme SimulPast [77], il y a un réel manque d'informations sur cette plateforme. L'implémentation ainsi que les mécanismes mis en œuvre ne sont pas beaucoup documentés. En particulier, nous n'avons pas trouvé d'informations sur la façon dont l'équilibrage de la charge est réalisé.

2.3 Evaluation des performances

L'objectif de paralléliser un modèle est soit d'obtenir de meilleures performances à l'exécution soit d'exécuter des modèles plus grands, ou plus précis, qui sont trop gros pour tenir dans la mémoire d'une seule machine. La question de la taille de la mémoire est adressée en ajoutant simplement plus de machines : leur mémoire est ajoutée au montant global. La question de la performance est traitée en donnant plus de processus à la simulation. Pour prouver l'efficacité du PDMAS dans ces deux problèmes, des expérimentations doivent être effectuées afin de mesurer les performances obtenues lors de l'exécution de plus grands modèles sur plusieurs processus. Nous avons exécuté notre modèle de référence sur les quatre PDMAS précédemment sélectionnés. Nous détaillons dans cette section les résultats de performance et nos observations lors de leurs exécutions sur des ressources HPC.

2.3.1 Paramètres expérimentaux

Pour l'évaluation des performances, nous utilisons le modèle de référence défini au Chapitre 1 et à la section 1.1.5 sur les quatre plateformes fonctionnelles c'est à dire : RepastHPC, D-Mason, Flame et Pandora. Au cours de l'implémentation du modèle, nous n'avons pas rencontré de difficultés notables. Nous avons exécuté le modèle de référence sur un cluster 1280 cœurs qui utilise un système de soumission batch SGE. Chaque nœud du cluster est un bi-processeurs, avec Xeon E5 (8 * 2 cœur). Les processeurs fonctionnent à la fréquence de 2.6 GHz avec 64 Go de mémoire, les nœuds sont reliés entre eux par un réseau DDR infiniband organisé en fat tree. Le système est partagé avec d'autres utilisateurs. Le système de traitement par lots garantit que chaque processus fonctionne sur son propre noyau sans le partager avec d'autres processus (pas de partage de temps). Lors de la soumission d'un job le nombre de cœurs demandés est donné à

l'ordonnanceur avec une option Round Robin. Cette option signifie que l'ordonnanceur tente de distribuer autant que possible les processus sur des nœuds différents. Ces réglages expérimentaux peuvent générer de petites variations dans le temps d'exécution de telle sorte que nous avons calculé ces variations sur les résultats de performance. La variation observée est faible, moins de 0,21% (obtenue avec Flame sur 128 cœurs). Pour cette raison, chaque mesure est prise 10 fois et nous faisons la moyenne de ces mesures.

A noter que cette configuration sera commune à toutes les expérimentations présentes dans la suite du manuscrit.

2.3.2 Analyse des résultats de performances

Bien que nous ayons été capables d'exécuter les quatre plateformes, D-Mason, Flame, Pandora, RepastHPC, sur une station de travail, seulement trois d'entre elles (RepastHPC, Flame et D-Mason) s'exécutent sur notre système HPC. En effet, les simulations de la plateforme Pandora ont un problème de deadlock sur notre système HPC même si nous utilisons les exemples fournis avec la plateforme. C'est pour cette raison que, la plateforme Pandora n'est pas inclue dans la comparaison des résultats de performances.

Lors de l'analyse de l'exécution des simulations de la plateforme D-Mason, nous avons remarqué que la plateforme utilise plusieurs threads par processus et que nous ne pouvons pas contrôler le nombre de threads lors de l'exécution des simulations. Les ordonnanceurs Batch comme SGE ou SLURM, et plus généralement les ordonnanceurs de lots utilisés dans les centres de calcul, supposent qu'un seul thread est exécuté par cœur alloué. Sinon les threads supplémentaires pourraient utiliser des cœurs qui sont attribués à d'autres utilisateurs ou exécutions. La plateforme D-Mason ne respecte donc pas les contraintes HPC et la comparer aux autres plateformes (RepastHPC et Flame) n'aurait pas de sens. En effet, cela reviendrait a comparés une exécution single-thread avec une exécution multi-threads, ce qui n'est pas comparable. En revanche, il est important de noter que nous constatons de bonnes performances pour la plateforme D-Mason. La plateforme D-Mason est une plateforme intéressante et à suivre lorsqu'elle aura atteint un plus grand niveau de maturité (il y a encore beaucoup de bugs et contraintes dûs à l'équilibrage de charges qui peuvent être améliorés). Pour cette raison, les résultats de performances présentés ne tiennent compte que des plateformes Flame et RepastHPC.

Nous avons réalisé plusieurs exécutions afin de présenter les comportements des plateformes concernant l'extensibilité (Figures 2.7), la montée en charge (Figure 2.8) et la consommation mémoire (Figure 2.9). Pour évaluer l'extensibilité nous faisons varier le nombre de nœuds utilisés pour l'exécution des simulations pendant que nous fixons le nombre d'agents. Pour tester la montée en charge, nous fixons le nombre de nœuds à 16, 32, 64 et 128 et nous faisons varier le nombre d'agents dans la simulation.

Les résultats d'exécutions pour l'extensibilité pour un modèle avec 200000 et 400000 agents sont donnés dans la Figure 2.7. Il est important de noter que la première exécution est réalisée avec 16 cœurs. Nous avons choisi 16 cœurs afin de pouvoir exécuter un maximum d'agents avec la limite mémoire de 4Go par cœurs. Nous pouvons noter que les deux plateformes passe bien à l'échelle c'est à dire qu'elles sont extensibles jusqu'à 64 cœurs, mais les performances ne progressent pas aussi bien lorsque plus de cœurs sont utilisés. Les résultats de la plateforme RepastHPC sont meilleurs que ceux de Flame : pour 16 cœurs, RepastHPC est 2,02 fois meilleur que Flame alors que pour 128 cœurs la différence est d'environ 9,26. Cette différence pourrait être expliquée par la stratégie utilisée par la plateforme pour la communication entre agents. Comme le modèle de référence est un modèle de communication et que Flame utilise les diffusions pour synchroniser les informations et communiquer, il est possible que la plateforme consomme beaucoup de temps dans le traitement des messages. En outre, on peut noter que, pour 400000 agents, la plateforme Flame est extensible jusqu'à 64 cœurs, mais avec davantage de cœurs celui

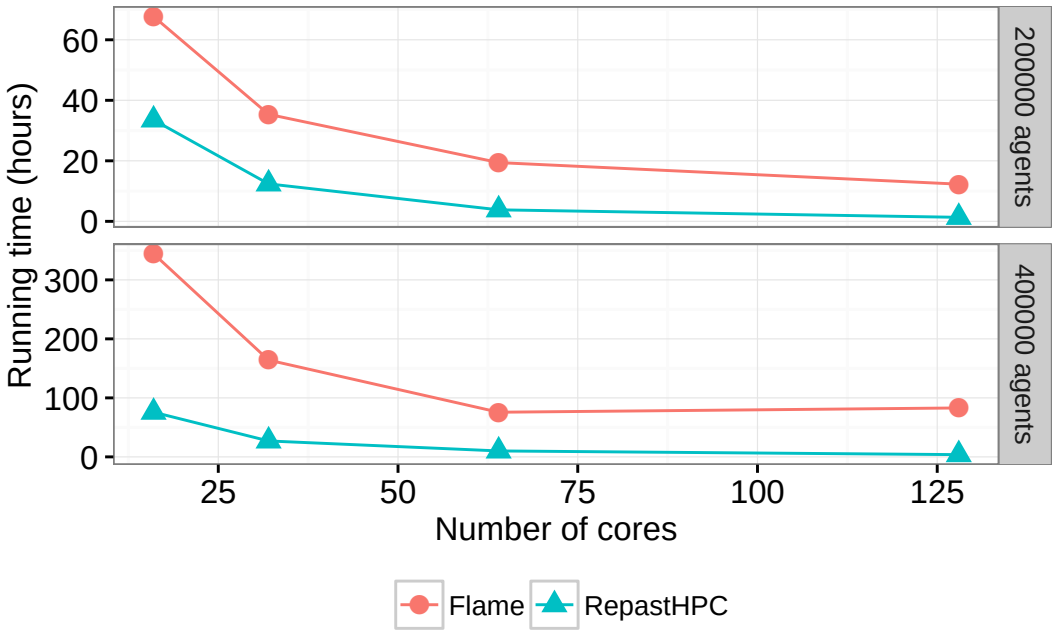


FIGURE 2.7 – Temps d'exécution pour les plateformes Flame et RepastHPC de 16 à 128 cœurs

conduit à une dégradation des performances.

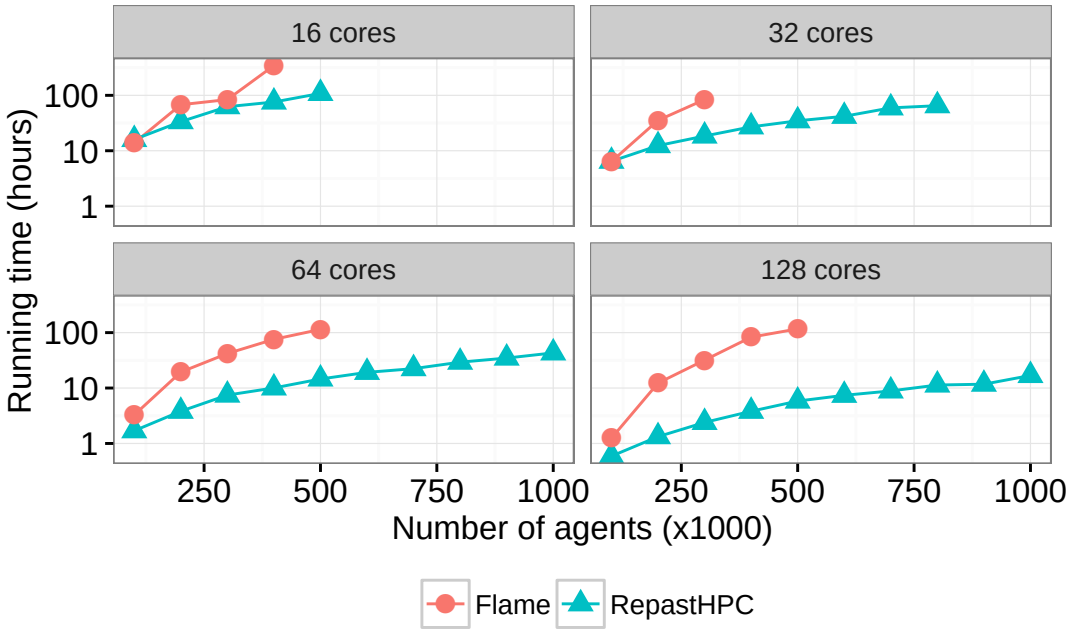


FIGURE 2.8 – Montée en charge des plateformes Flame et RepastHPC

La Figure 2.8 présente les performances de montée en charge des deux plateformes pour 16, 32, 64 et 128 cœurs. La montée en charge est obtenue en fixant la charge interne des agents (taille du calcul de DFT fixé à 128), et en augmentant le nombre d'agents dans la simulation.

Les graphiques ne montrent pas de résultats pour toutes les valeurs de l'axe X car les plateformes ne fonctionnent pas avec autant d'agents sur ce nombre de cœurs. Nous pouvons voir que Flame ne passe pas aussi bien à l'échelle que RepastHPC. Par exemple, pour 128 cœurs avec Flame, nous n'avons pas été en mesure d'exécuter des simulations de plus 500000 agents.

La Figure 2.8 montre également que la plateforme RepastHPC réagit vraiment mieux à la montée en charge que la plateforme Flame. Il est évident que le modèle utilisé n'utilise pas toute la puissance de la plateforme Flame étant donné que la plateforme est limitée en terme de communications inter-agents. Néanmoins, sur la Figure 2.8, pour 16 cœurs et 100000 agents, nous pouvons noter que la plateforme Flame offre de meilleurs résultats que RepastHPC. La question est donc : est-ce dû à l'utilisation du concept X-Machines ou en raison des mécanismes de synchronisation ? Une autre explication possible est que la plateforme Flame gère la synchronisation des agents distants alors que cela n'est pas géré dans RepastHPC.

Il est à noter que seul RepastHPC atteint 1 millions d'agents sans aucun problème et sans consommer beaucoup de mémoire.

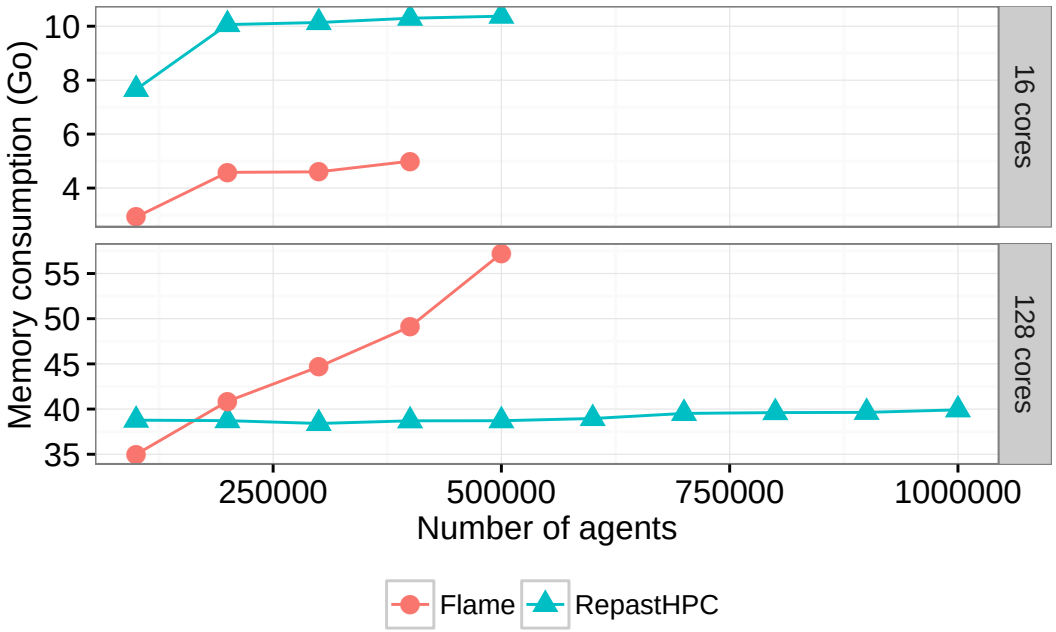


FIGURE 2.9 – Consommation mémoire des plateformes Flame et RepastHPC

Pour finir, la Figure 2.9 représente la consommation de mémoire de plateformes lors des exécutions des simulations pour 16 et 128 cœurs. Sur la Figure 2.9 pour 16 cœurs, nous pouvons remarquer que Flame utilise environ 2 fois moins de mémoire que RepastHPC. A l'inverse, pour 128 cœurs, il y a un écart entre la consommation de mémoire des deux plateformes. Alors que RepastHPC reste à peu près constant en consommation mémoire, Flame augmente rapidement sa consommation mémoire.La plateforme Flame utilise plus de 57 Go de mémoire pour exécuter une simulation sur 128 cœurs alors RepastHPC utilise environ 39 Go de mémoire pour la même simulation. Ces résultats pourraient être expliqués par la mise en œuvre interne des agents dans la plateforme mais aussi par la façon de gérer les communications et les synchronisations au cours de la simulation.

2.4 Limites des plateformes existantes

Dans cette section, nous donnons notre retour d'expérience sur les plateformes testées. Cette expérience a été acquise au travers des travaux bibliographiques, de la découverte des plateformes, de la mise en œuvre de différents modèles et des expérimentations. Cette analyse représente environ une demi-année de travail accumulée au cours des deux dernières années et plus de 250.000 heures de calcul. Nous résumons nos impressions et notre avis informel sur les points positifs et négatifs de chaque plateforme.

RepastHPC

RepastHPC est une plateforme complète qui permet l'utilisation de différentes structures (réseaux, grille). Ces structures peuvent être couplées dans une même simulation pour représenter les différents types d'interactions. Le développement est facilité par un grand nombre de tutoriels qui expliquent comment utiliser RepastHPC. Ces tutoriels sont clairs et détaillés. RepastHPC propose deux façons d'implémenter des simulations multi-agents : ReLogo et C++. L'utilisation de ReLogo pour développer un modèle est très simple, mais le langage ne bénéficie pas de toutes les fonctionnalités de RepastHPC. D'autre part en utilisant C++ cela permet d'implémenter des simulations plus complexes avec pour seul inconvénient que de bonnes compétences en C++ (notion de modèles) sont nécessaires.

Pour-	Contre-
— Documentation et exemples compréhensifs — Facilité de configurations et flexibilité de l'ordonnanceur : paramétrable pour effectuer des ordonnancements complexes — Ne nécessite pas de compétences spécifiques en MPI — Grande communauté réactive — Facilité de configuration et d'exécution sur un cluster — Régulièrement mis à jour	— N'inclue pas tous les cas de communications entre agents — Ne synchronise pas les informations lorsqu'une copie d'agent est modifiée

Table 2.4 – Arguments Pour- et Contre- de la plateforme RepastHPC

Il y a cependant quelques limites dans l'utilisation de la plateforme car aucune communication n'est autorisée entre les agents distants. Cette absence de communications distantes ne permet pas d'implémenter certains modèles agent, même les plus simples, qui ont besoin de modifications à distance des agents. RepastHPC offre de bonnes performances et de l'efficacité uniquement avec des modèles en lecture.

Après l'implémentation de notre modèle sur la plateforme Repast-HPC, nous avons synthétisé notre retour d'expérience sur cette plateforme. Les arguments pour- et contre- sont donnés dans la Table 2.4.

Flame

Flame est une plateforme intéressante pour développer des modèles parallèles car elle utilise une approche différente par rapport aux autres plateformes, (X-Machines ou graphes d'état).

Bien que cette approche implique d'apprendre un nouveau langage de programmation, il se trouve qu'il n'est pas plus complexe à mettre en œuvre une simulation dans la plateforme Flame que sur une autre plateforme. En outre, un grand nombre d'exemples sont proposés ce qui facilite l'apprentissage de l'implémentation d'une simulation en utilisant le langage XMML. L'utilisation de X-Machine permet de cacher les problèmes de distribution au développeur et d'exécuter de manière transparente des simulations sur un super-calculateur.

Pour-	Contre-
— Facilité de développement une fois la syntaxe XMML connue — Orienté sur l'aspect état-transition — Possibilité de jouer graphiquement la simulation après exécution (optionnel mais apprécié) — Méta-model XMML pour valider la simulation — Génération de code parallèle automatique — Facilité pour passer d'une version exécuté en séquentielle à une version exécuté en parallèle	— Difficulté pour comprendre le modèle de simulation XMML — Impossible d'envoyer et de recevoir un message dans le même état — Pas de communication point-à-point, toutes les communications doivent utiliser les tableaux de messages

Table 2.5 – Arguments Pour- et Contre- de la plateforme Flame

Contrairement à la plateforme RepastHPC, la plateforme Flame permet la modification distante d'agents par l'intermédiaire de messages et des tableaux de messages. Cette fonctionnalité est acquise au prix de performances dégradées, même pour les modèles en lecture seule. Deux choix dans la conception de la plateforme Flame permettent d'expliquer ses performances : l'absence d'optimisation de la zone de recouvrement et l'utilisation systématique de diffusion pour les communications. On peut noter qu'une nouvelle version de la plateforme Flame qui résout le problème de performance est annoncée [78] mais n'est malheureusement pas encore disponible.

La Table 2.5 regroupe les arguments pour- et contre- le développement d'un modèle avec la plateforme Flame.

D-Mason

Par rapport aux plateformes RepastHPC et Flame, la plateforme D-Mason est plus récente et cible davantage les réseaux de stations que les super-calculateurs. Elle est cependant une plateforme riche en fonctionnalités qui propose différents moyens de communications comme ActiveMQ ou MPI.

Il est très facile de développer une simulation avec D-Mason car toutes les méthodes de base sont déjà implémentées. Il n'est pas nécessaire d'avoir des compétences ou des connaissances spéciales en langage Java pour développer un modèle agent. D-Mason est une grande plateforme, cependant il lui manque peut-être un peu de maturité pour atteindre les clusters HPC. Nous ne sommes malheureusement pas parvenus à l'exécution d'une version stable de la plateforme basée MPI avec notre modèle de référence. Notez que, comme RepastHPC, D-Mason ne permet pas la modification des agents distants et se limite donc aux modèles en lecture seule.

La plateforme D-Mason possède une documentation détaillée qui explique comment transformer un modèle Mason en un modèle parallèle pour D-Mason. Il existe aussi de nombreux

Pour-	Contre-
— Simple et intuitif (interface graphique) — Facilité de compilation — Communauté active — Régulièrement mis à jour — Propose deux couches de communications (MPI or ActiveMQ JMS) — Propose trois manières de synchronisations basée MPI (BCast, Gather and Parallel)	— N'inclue pas tous les cas de communications entre agents

TABLE 2.6 – Arguments Pour- et Contre- de la plateforme D-Mason

exemples de modèles dans la plateforme D-Mason qui aident à la compréhension des fonctionnalités de la plateforme.

Après l'implémentation de notre modèle dans la plateforme D-Mason, nous avons synthétisé notre ressenti avec les pour- et contre- dans le tableau 2.6.

Pandora

Comme les plateformes précédentes, Pandora est une plateforme complète permettant d'implémenter des modèles agents parallèles. Une fonctionnalité très appréciable fournie par la plateforme est que le code parallèle est généré automatiquement. Nous ne sommes pas parvenus à l'exécution d'un modèle avec Pandora sur le super-calculateur, mais nous l'avons fait sur une station de travail. Néanmoins, pour comprendre comment développer une simulation avec Pandora d'autres documents ou tutoriels seraient appréciables au lieu de lire des exemples de code. Pandora intègre nativement le soutien SIG, et le code C++ d'une simulation est très simple à écrire.

La plateforme Pandora ne fournit pas la communication distante, mais, grâce à son modèle de synchronisation tournant, elle résout le problème des modèles en lecture-écriture. Ce modèle de synchronisation limite cependant l'environnement de modèle à deux dimensions et il ne prend pas en charge, pour autant que nous le comprenons, d'autres structures que les grille, pour représenter l'environnement (pas de graphe).

La Table 2.7 résume les arguments pour- contre- pour le développement d'un modèle avec la plateforme Pandora.

Après cette comparaison détaillée des plateformes multi-agents parallèles existantes, ce que nous pouvons mettre en avant est que la plupart des plateformes proposent des mécanismes permettant l'exécution des simulations agents parallèles mais avec certaines limites. Par exemple, la gestion des synchronisations entre les comportements des agents est souvent limitée à la lecture. En ce qui concerne la communication entre les agents, nous constatons le même problème, à savoir la communication est uniquement limitée aux agents s'exécutant sur le même processus. Pour finir, la décomposition de la simulation en portions qui sont distribuées est quasiment toujours basée sur un découpage cartésien ce qui peut ne pas être le plus efficace dans tous les cas.

Par conséquent, pour répondre à ces limites nous présentons nos contributions dans les chapitres suivant.

Pour-	Contre-
— Génération de code parallèle automatique — Support SIG — Facilité de développement de modèle — Facilité pour passer d'une version exécuté en séquentielle à une version exécuté en parallèle — Ne nécessite pas de compétences spécifiques en MPI — Outil d'analyse avancée (Cassandra)	— Trop peu de documentations — N'inclue pas tous les cas de communications entre agents — N'inclue pas tous les cas de synchronisations entre agents

TABLE 2.7 – Arguments Pour- et Contre- de la plateforme Pandora

Deuxième partie

FractalPMAS : une approche de parallélisation des Systèmes Multi-Agents à base de *Nested Graphs*

Chapitre 3

Distribuer les Systèmes Multi-Agents avec les *Nested Graphs*

Sommaire

La deuxième partie de ce mémoire se décompose en trois chapitres. Le premier chapitre 3 est consacré à la manière de modéliser et de distribuer les systèmes multi-agents à l'aide de *Nested Graphs*. Ensuite, dans le chapitre 4 nous nous intéressons à la synchronisation dans les systèmes multi-agents parallèles. Nous étudions l'impact de la synchronisation sur les résultats mais aussi sur le temps d'exécutions des simulations. Pour finir, le chapitre 5 concerne la manière de gérer les communications entre agents, nous proposons un schéma de communication qui permet de communiquer entre tous les agents de la simulation.

Actuellement, les PDMAS utilisent majoritairement les grilles pour représenter l'environnement dans lequel les agents évoluent. En effet, cette structure de données répond bien aux attentes des modélisateurs pour les modèles où les agents sont situés dans un espace. De plus, la distribution d'une grille sur un ensemble de cœurs permet de bénéficier d'une plus grande puissance de calcul et de plus de mémoire. En revanche, le découpage de la simulation n'est pas forcement efficace comme nous allons le montrer. C'est la raison pour laquelle, nous nous sommes intéressés à la manière de modéliser une simulation multi-agents afin de faciliter sa distribution sur les processus en tirant parti des méthodes ou outils existants et utilisés dans le domaine du parallélisme.

Dans la section suivante, nous montrons les limites inhérentes aux grilles lorsqu'elles doivent être distribuées, puis nous détaillons la réflexion qui nous a conduit à proposer d'autres structures de données. Nous définissons un formalisme de modélisation que nous expliquons à l'aide de l'exemple du modèle proie-prédateur précédemment défini dans la section 1.1.5. Nous abordons ensuite l'application de notre formalisme dans la distribution des simulations multi-agents, ce que nous illustrons à la fois à travers l'implémentation de modèles dans la plateforme FractalPMAS et l'étude des performances de ces implémentations, pour valider l'intérêt et l'applicabilité de la proposition.

3.1 Limites de la distribution à base de grilles

Les grilles sont une bonne base pour représenter un environnement en deux dimensions sur lequel les agents peuvent se déplacer, comme en témoigne le nombre important de modèles multi-agents basés sur les grilles ou le fait que certains simulateurs, comme Netlogo par exemple, utilisent implicitement cette structure de données pour représenter les environnements.

Pour distribuer cette grille, les PDMAS utilisent la décomposition cartésienne comme nous l'avons présenté au chapitre 2. Cette décomposition permet de distribuer l'environnement sur un axe, l'axe x des abscisses, ou sur deux axes x et y comme montré dans les Figures 3.1 et 3.2. Nous obtenons alors une distribution en bandes, dans le cas d'une distribution sur un seul axe, ou une distribution carrée ou rectangulaire, dans le cas d'une distribution sur deux axes.

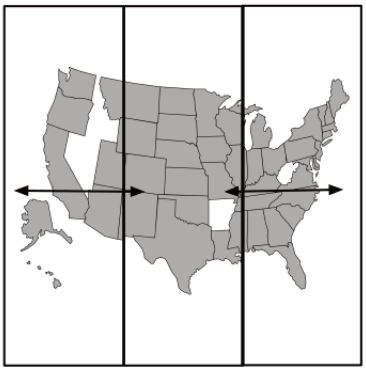


FIGURE 3.1 – Un exemple de décomposition de grille sur l'axe x avec la plateforme D-MASON sur trois processus [2]

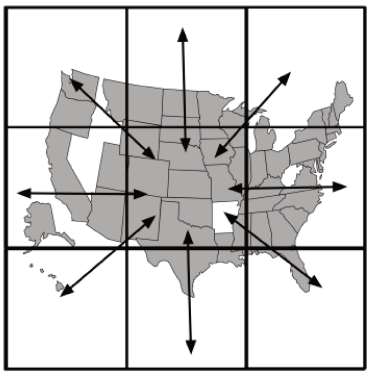


FIGURE 3.2 – Un exemple de décomposition de grille sur les axes x et y avec la plateforme D-MASON sur neuf processus [2]

Une fonction d'affectation réalise ensuite la correspondance entre les groupes de cellules (bande ou rectangle) et les cœurs de calcul. Cette fonction affecte l'ensemble des cellules d'un groupe à un cœur. Malgré une division à grain fin, avec des cellules de petite taille, comme cela est fait dans la plateforme D-MASON [79], cette affectation régulière de la simulation peut ne pas être appropriée pour tous les types de modèles même si la répartition de charge est dynamique car la division sera toujours rectiligne. En effet, dans les SMA, la charge est principalement générée par l'exécution des comportements des agents et non par l'environnement représenté par la grille qui encapsule en fait les données. Cette structure de grille peut alors ne pas s'avérer assez flexible, pour équilibrer correctement la charge générée par les cellules sur les différents processus participant à la simulation. En effet, le grain d'affectation est une bande de cellules, ce qui peut être trop grossier pour un équilibrage fin.

Comme nous travaillons sur les PDMAS animés par pas de temps, cela implique que tous les processus sont exécutés de manière synchrone et que le processus le plus lent va déterminer la

vitesse d'exécution de la simulation. De ce fait, il est particulièrement important de correctement distribuer et équilibrer la charge de la simulation car cela a un impact direct sur les performances.

Nous pouvons illustrer le manque de flexibilité de la structure de grille dans la distribution des cellules à l'aide du modèle proie-prédateur. Par exemple, la Figure 3.3 représente une configuration initiale arbitraire de ce modèle. Cette configuration est basée sur une structure de grille de taille 3x3, pour représenter l'environnement, sur laquelle les agents (loups, moutons) peuvent évoluer.

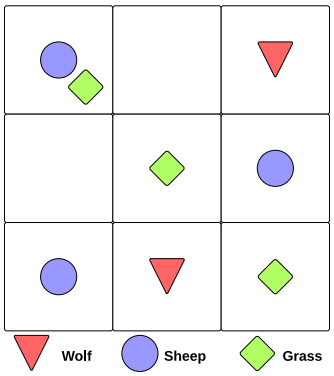


FIGURE 3.3 – Configurations initiales du modèles proie-prédateur

Si nous distribuons cette configuration initiale sur deux processus en utilisant une grille comme c'est généralement implémenté dans les PDMAS, nous pouvons obtenir une des deux décompositions présentées sur la Figure 3.4. Nous constatons sur cette Figure 3.4 que la bande du milieu ne peut être découpée pour être répartie entre les deux processus. La distribution basée sur la grille est donc contrainte par la structure de grille elle même et la décomposition utilisant un seul axe (x) ne peut pas balancer correctement la densité des agents sur chacun des processus. Nous pouvons noter que le problème est symétrique si nous effectuons une distribution seulement sur l'axe des ordonnées y.

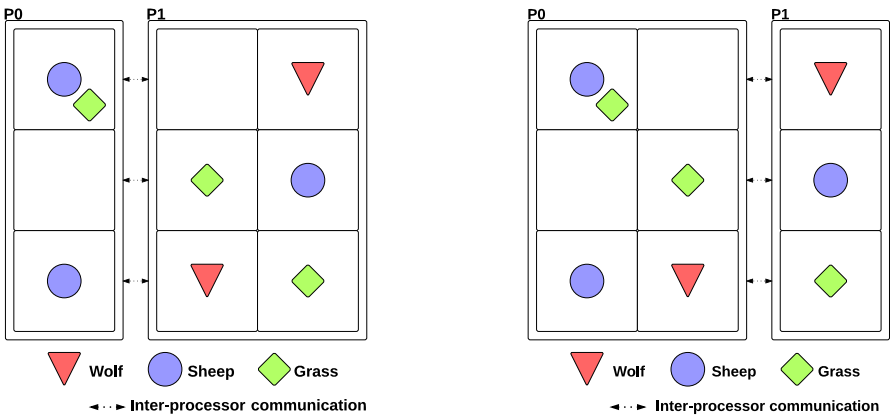


FIGURE 3.4 – Exemples de décomposition de grille sur l'axe x pour le modèle proie-prédateur utilisant 2 processus

Si nous distribuons la même simulation sur quatre processus, le déséquilibre de charge est davantage prononcé. La Figure 3.5 représente un exemple de distribution du même modèle mais maintenant sur quatre processus. Il est clair, sur le schéma, que le découpage ne peut être modifié qu'en déplaçant des cellules sur l'un ou sur l'autre des axes, engendrant à chaque fois

le déplacement d'une bande complète de cellules et impliquant cette fois plusieurs processus. Au regard de la densité d'agents dans les partitions distribuées sur chacun des processus, nous constatons qu'ils ne possèdent pas la même charge.

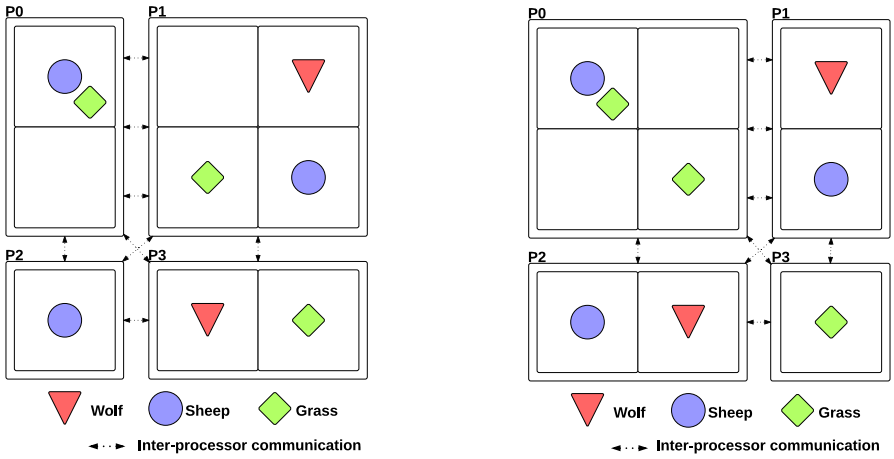


FIGURE 3.5 – Exemples de décomposition de grille sur les axes x et y pour le modèle proie-prédateur sur quatre processus

Évidemment les configurations présentées sont trop simples pour nécessiter une parallélisation mais elles sont utilisées à titre d'exemple pour présenter comment un environnement basé sur une structure de grille peut être distribué sur un ou deux axes. En utilisant ces configuration simple, nous souhaitons montrer les difficultés qu'impliquent la distribution d'une grille où les agents ne sont pas uniformément distribués. Bien entendu, le nombre d'agent de type *Cell* n'a pas besoin d'être identique sur chacune des partitions distribuées sur les processus. Par exemple, les Figures 3.4 et 3.5, montrent que les décompositions ne sont pas uniformes en terme de nombre d'agents de type *Cell* distribués sur chaque processus ni même en terme d'arcs coupés. Il est important de noter que les décompositions sont limitées car la structure de grille limite le découpage à des lignes ou des colonnes entières. Par conséquent, même avec une fine granularité de la décomposition, nous ne pouvons pas atteindre un équilibrage fin de la charge. Il est d'ailleurs difficile d'adapter dynamiquement cette décomposition lorsque la charge varie et implique plusieurs processus.

A noter que, même si cela n'est pas implémenté dans les PDMAS, une distribution sur trois axes ou plus est envisageable. Elle n'apporterait cependant pas de souplesse supplémentaire dans la distribution.

3.2 Proposition

Pour pallier le manque de flexibilité de la structure de la grille, nous proposons d'utiliser les graphes comme base pour modéliser les systèmes multi-agents. Les graphes sont des structures largement utilisée dans le domaine de l'informatique distribuée ou parallèle afin de modéliser des problèmes d'ordonnancement [80] ou encore de partitionnement [81]. L'un des avantages de cette structure de graphe est qu'il existe de nombreux outils pour partitionner ou distribuer des graphes pour les machines parallèles tels que les bibliothèques Parmetis [82], PT-Scotch [83] ou encore Zoltan [84]. Ces bibliothèques offrent de bons résultats de performances pour des graphes de grande taille [85]. C'est la raison pour laquelle nous souhaitions tirer parti de ces outils afin d'améliorer l'exécution des PDMAS dans un environnement parallèle. Pour bénéficier

de la puissance de ces bibliothèques, nous devons changer la manière dont les modèles agents sont conçus, représentés et exécutés.

Habituellement, une simulation multi-agents peut être vue comme un ensemble d'entités interagissant entre elles mais aussi avec l'environnement dans lequel elles évoluent. L'environnement peut lui aussi être considéré comme un ensemble d'agents [86]. Il est donc possible d'imaginer représenter les modèles multi-agents en utilisant des graphes dans lesquels les nœuds représentent les agents et les arêtes représentent les interactions ou un quelconque lien entre les agents. Cependant, le travail de modélisation peut s'avérer complexe et fastidieux car la structure de graphe n'est pas toujours facile et naturelle à utiliser pour les utilisateurs de systèmes multi-agents. En revanche, si la simulation agent est représentée sous forme de graphe, il est alors plus aisé de proposer des algorithmes ou des outils afin de la distribuer ou encore d'effectuer de la répartition de charge. C'est donc un compromis entre difficulté de modélisation et utilisation de bibliothèques de partitionnement existantes qu'il faut effectuer. C'est la raison pour laquelle nous proposons d'utiliser une structure basée sur les *Nested Graphs*, pour modéliser les simulations multi-agents. Cette structure de *Nested Graphs* intègre nativement une manière plus facile et plus flexible d'utiliser la structure de graphes tout en facilitant la distribution ainsi que l'équilibrage de charge pour les environnements parallèles.

Notre proposition est basée sur l'utilisation des *Nested Graphs* et de transformation de *Nested Graphs* comme moyen de modéliser des modèles multi-agents multi-niveaux, à large échelle. Les *Nested Graphs* sont des graphes dont les nœuds peuvent eux mêmes être des *Nested Graphs*. C'est donc une structure de donnée récursive. Le nouvel aspect introduit par les *Nested Graphs* est la définition de différents niveaux d'abstractions qui peuvent être utilisés comme un moyen conceptuel pour diviser ou structurer un modèle de manière hiérarchique. Dans [87], les auteurs introduisent un modèle de *Nested Graphs* pour représenter et manipuler des structures complexes appliqués aux bases de données. Dans le contexte des système multi-agents les *Nested Graphs* ont déjà été utilisés mais pas dans le contexte des systèmes multi-agents parallèles. En effet, dans [88], les auteurs utilisent les *Nested Graphs* pour représenter les interactions dans les modèles complexes. Utiliser une structure hiérarchique est une manière de conceptualiser plus facilement comment un système complexe est modélisé. Dans la suite de ce chapitre, nous montrons que l'utilisation des *Nested Graphs* permet la description de tout système multi-agents à différents niveaux d'abstractions mais aussi que les *Nested Graphs* sont nativement conçus pour être distribués de manière efficace dans un environnement parallèle.

3.3 Description formelle

Nous basons notre proposition sur deux principes définis dans [88] qui formalisent une simulation multi-agents :

— tout agent peut dynamiquement encapsuler un environnement. C'est la base d'une structure imbriquée récursive, mais cette structure doit aussi être en mesure de changer au cours du temps,

— tout agent peut être situé dans plusieurs environnements en même temps, sans avoir une idée préalable de ce que ces environnements représentent (un niveau micro / macro du monde physique, un groupe, une organisation, une mémoire spatiale, un réseau social, etc.).

Dans notre cas, nous modifions le premier principe afin d'adapter la définition à notre proposition :

— Tout agent peut dynamiquement encapsuler un **agent**. C'est la base d'une structure imbriquée récursive, mais cette structure doit aussi être en mesure de changer au cours du

temps.

Dans notre proposition, tous les composants ou entités qui participent à la simulation sont des agents comme le préconisent les auteurs dans [86]. Cela signifie que l'environnement lui même est modélisé par un ou plusieurs agents, suivant le type d'environnement nécessaire à la simulation. Ainsi avec notre formalisme les agents se définissent de la manière suivante :

— chaque agent dans la simulation est un *Nested Graph* typé et labellisé que nous appelons *Agent Graph*,

— les comportements des agents sont représentés par des transformations d'*Agent Graph*, c'est à dire des modifications d'arc et de nœuds dans le graphe,

— les relations entre les agents appartenant à un même contexte, c'est à dire contenus dans un même *Agent Graph*, sont représentées par un ou plusieurs arcs typés et valués. Ces relations peuvent représenter des communications entre agents ou encore une position relative par rapport à un environnement.

Formellement, soit Γ un ensemble de types, Σ un ensemble de labels et Λ un ensemble de valeurs. Nous définissons $\mathcal{G}$ un graphe agent appartenant à $\mathbb{G}$ l'ensemble des graphes agents récursivement de la manière suivante :

$$\mathcal{G} \in \mathbb{G} \Leftrightarrow \mathcal{G} = \langle \mathfrak{G}, \mathfrak{E}, \mathfrak{T}, \mathfrak{L}, \mathfrak{V} \rangle \tag{3.1}$$

où $\mathfrak{G} \subseteq \mathbb{G}$ est un ensemble d'*Agent Graphs* (aussi appelé nœuds) représentant les agents du modèle, $\mathfrak{A} \subseteq \mathfrak{G} \times \mathfrak{G}$ est un ensemble d'arcs orientés représentant les interactions ou la structure entre les agents, $\mathfrak{T} : \mathfrak{G} \mapsto \Gamma$ est une fonction de typage qui assigne un type à chacun des agents, $\mathfrak{L} : \mathfrak{G} \mapsto \Sigma$ est une fonction de labellisation qui assigne un label à chacun des agents et $\mathfrak{V} : \mathfrak{A} \mapsto \Lambda$ est une fonction permettant de valuer les arcs en leur assignant une valeur.

L'état de la simulation est donc entièrement décrit par les *Agent Graphs* (équation 3.1) qui la composent. Dans cette méthode de modélisation, les comportements sont décrits par des transformations d'*Agent Graph* qui sont représentées par une paire d'*Agent Graphs*, nommées *AG_Pre* et *AG_Post* qui respectivement représentent les états avant et après l'exécution de la transformation. Chaque *Agent Graph* contient un nœud mis en évidence (en gras) comme sur la Figure 3.6. Ce nœud en gras représente le nœud qui initie et exécute la transformation. Cette notation en gras est nécessaire au bon fonctionnement de la méthode de modélisation et à la prise en compte de tous les cas de modélisation de comportements. De cette manière, si un comportement ne peut s'exécuter que si plusieurs nœuds du même type sont présents alors il pourra être modélisé. Dans le cadre d'une transformation, c'est à dire l'application d'un comportement, si *AG_Pre* est trouvé comme un sous *Agent Graph* composant la simulation (correspondance identique entre *AG_Pre* et l'état de la simulation) alors le sous *Agent Graph* de la simulation est transformé en *AG_Post*. En d'autres termes, l'*AG_Pre* de la transformation d'*Agent Graph* peut être vu comme un schéma, un motif qui doit être reconnu dans la simulation avant de pouvoir modifier la simulation en l'*AG_Post*. Nous pouvons faire une analogie avec les structures conditionnelles (*if* [le schéma correspondant à *AG_Pre* est reconnu dans la simulation] *then* [transformation d'*AG_Pre* pour qu'il coïncide avec l'*AG_Post*]). De cette manière le comportement souhaité peut être réalisé.

En se basant sur ces définitions, notre proposition repose sur deux points importants :

— Une méthode de modélisation où les systèmes multi-agents sont et peuvent être modélisés sous forme graphique. Dans cette modélisation, tous les éléments du modèle multi-agents sont des agents sans différence entre l'environnement et les agents. Les agents et leurs relations sont représentés par des *Agent Graphs*, une représentation de graphes imbriqués. Les comportements des agents quant à eux, sont définis sur la base de transformations *Agent Graph*.

— Un PDMAS adapté utilisant des modèles d'*Agent Graphs* est utilisé pour exécuter efficacement des simulations dans un environnement parallèle.

Avec cette proposition, les SMA peuvent donc être modélisés d'une manière graphique tout en bénéficiant d'une base formelle similairement aux réseaux de pétri [89]. Cette modélisation intègre aussi les différents niveaux d'abstractions nécessaires pour faciliter le travail du modélisateur. Pour illustrer, notre proposition, nous utilisons le modèle agent classique Proie-Prédateur défini en section 1.1.5.

3.4 Illustration

Dans un premier temps, nous introduisons quelques notations graphiques afin de permettre au lecteur une meilleure compréhension de notre méthode et dans un deuxième temps, nous appliquons notre proposition au modèle Proie-Prédateur défini en section 1.1.5.

3.4.1 Notations graphiques

Un nœud *Agent Graph* (Figure 3.6) est représenté par une ellipse à laquelle est associée un label, donné sous la forme *Label :Type* ($\mathfrak{L}$,$\mathfrak{T}$). Ce label et ce type permettent de caractériser les nœuds associés. Les arcs, qui représentent les relations entre les nœuds ($\mathfrak{E}$), sont représentés par des flèches labellisées sous la forme *Type :Value* ($\mathfrak{T}$, $\mathfrak{V}$) ce qui permet d'associer un type et une valeur à un arc. Pour finir, chaque *Agent Graph* contient un nœud de type *origin*. Ce nœud permet de lier à l'aide d'arc les agents contenus dans cet *Agent Graph* et donc de permettre la création de hiérarchies.

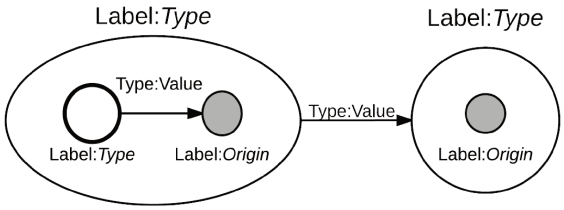


FIGURE 3.6 – Représentation générique d'*Agent Graphs*

Une transformation d'*Agent Graph* est décrite à l'aide de deux *Agent Graphs* reliés par une double flèche comme présenté en Figure 3.7. Pour rappel, l'ellipse en gras représente le nœud qui initie la transformation d'*Agent Graphs*.

3.4.2 Illustration de la modélisation avec le modèle proie-prédateur

La Figure 3.8 présente les niveaux d'abstraction pour le modèle proie-prédateur. L'environnement du modèle proie-prédateur est une grille de cellules comme énoncé dans la définition des modèles en section 1.1.5. La grille représente le plus haut niveau d'abstraction de la simulation (niveau 1). Dans le cadre de notre méthode, cette grille est modélisée à l'aide d'*Agent Graph* de type *Cell*. Chaque *Agent Graph* de type *Cell* est lié aux autres nœuds qui lui sont adjacents à l'aide d'un arc de type *adjacent* afin de représenter une structure de grille comme présenté dans la Figure 3.9.

Les *Agent Graphs* de type *Cell* contiennent les *Agent Graphs* de type herbe, mouton et loup. Chaque *Agent Graphs* de type herbe, mouton et loup est lié à l'aide d'un arc labelisé '*on*' au nœud *origin* de l'agent de type *Cell* dans lequel il est situé.

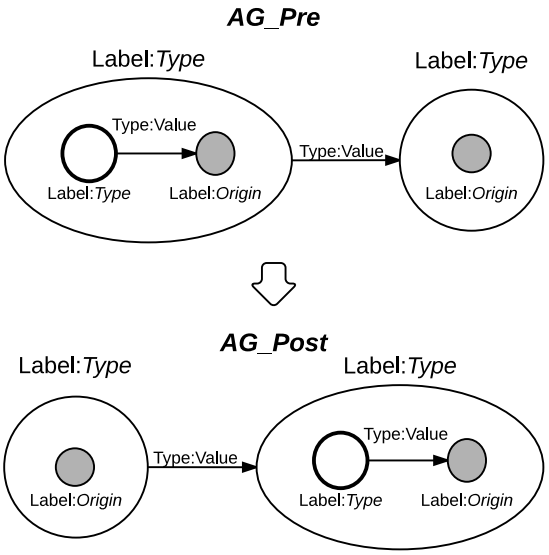


FIGURE 3.7 – Représentation générique d'une transformation d'*Agent Graphs* (Déplacement - Move)

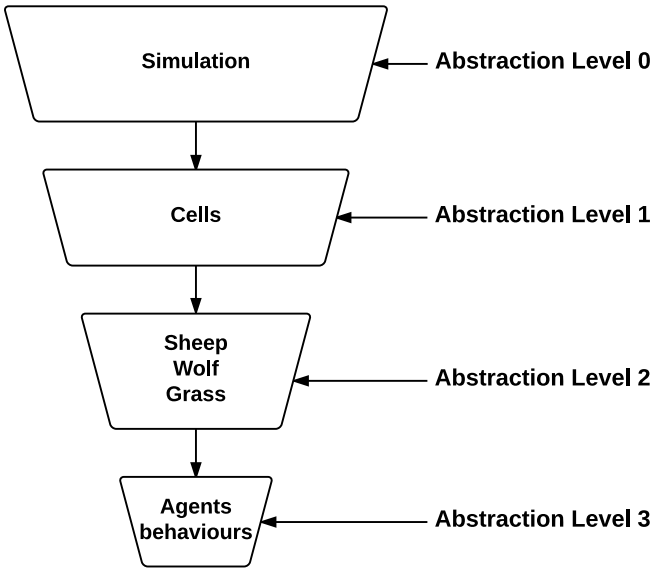


FIGURE 3.8 – Schéma des niveaux d'abstraction pour le modèle proie-prédateur

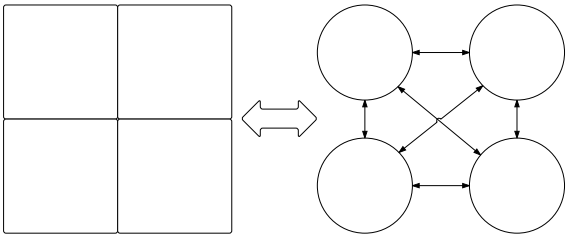


FIGURE 3.9 – Représentation d'une grille à l'aide de graphe

Le troisième niveau d'abstraction représente les caractéristiques des agents contenus dans les agents de type *Cell*. Ces agents quelque soit leur type (mouton, loup ou herbe) possèdent aussi un nœud *origin* auquel est rattaché un arc récursif (de l'origine à l'origine) de type force qui représente la force vitale (ou énergie de l'agent). Un nœud représentant un agent de type herbe ne possède pas de caractéristique particulière il reste donc vide.

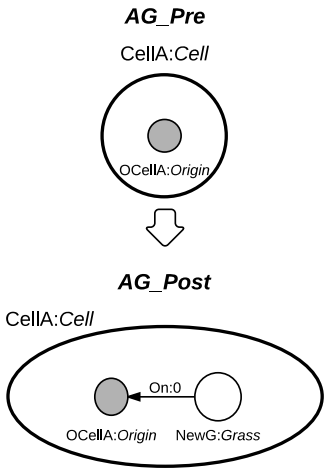


FIGURE 3.10 – Représentation du comportement *Grass growth*

A chaque pas de temps de la simulation, les comportements suivants, modélisés à l'aide de transformations d'*Agent Graph* sont appliqués :

— Le comportement Grass growth (Figure 3.10) : à l'aide d'une probabilité donnée par le modélisateur l'herbe a la possibilité de repousser tous les 'n' pas de temps.

— Le comportement "Move" (Figure 3.11(a)) : les moutons ou les loups peuvent se déplacer aléatoirement sur une cellule adjacente à celle où ils se situent.

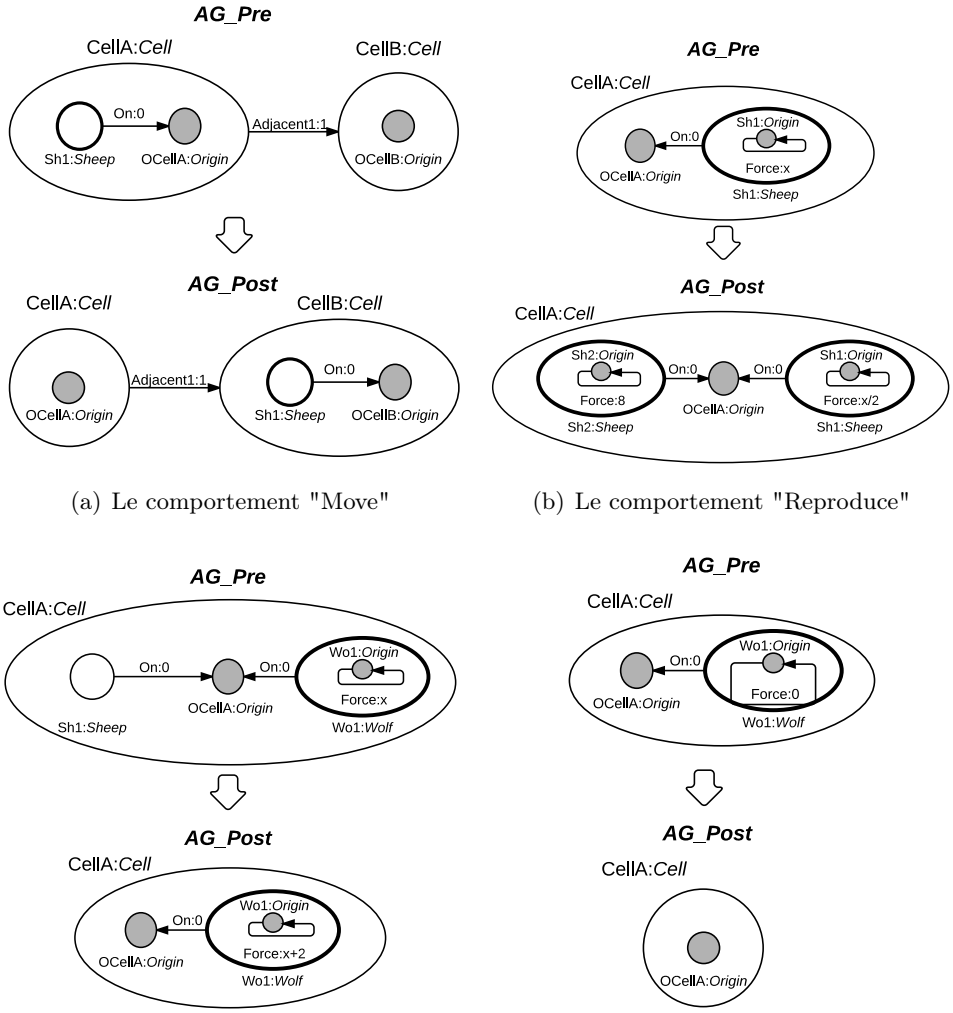
— Le comportement "Eat" (Figure 3.11(c)) : Les moutons (ou les loups) mangent de l'herbe, cela a pour action d'augmenter leur force ou énergie. Une fois que l'herbe a été mangée, elle doit disparaitre de la simulation.

— Le comportement "Reproduce" (Figure 3.11(b)) : à l'aide de la probabilité définie par le modélisateur les loups ou les moutons ont la capacité de se reproduire, et donc de créer un nouveau mouton ou loup avec une force qui est celle de son créateur divisée par deux. Le créateur, lui aussi, voit sa force divisée par deux.

— Le comportement "Die" (Figure 3.11(d)) : les moutons ou respectivement les loups meurent lorsque leur énergie est inférieure à zéro. Dans ce cas ils disparaissent de la simulation.

La Figure 3.12 présente le workflow qui définit l'ordre dans lequel les comportements des agents moutons sont appliqués à chaque pas de temps. Ce workflow est pratiquement identique pour les loups, à l'exception du comportement "Eat" puisque les loups ne mangent pas d'herbe mais des moutons.

L'utilisation des *Nested Graphs* pour modéliser les simulations multi-agents permet de représenter graphiquement les agents ainsi que leurs comportements sur plusieurs niveaux. Cependant, ce n'est pas le seul avantage de cette structure. En effet, nous présentons dans la suite, l'autre avantage de cette structure de graphes qui est de fournir une base adaptée à la distribution des modèles. Pour cela, nous présentons la plateforme FPMAS qui permet d'implémenter et d'exécuter un modèle agent parallèle à l'aide du formalisme que nous venons de définir.



(a) Le comportement "Move"

(b) Le comportement "Reproduce"

(c) Le comportement "Eat"

(d) Le comportement "Die"

FIGURE 3.11 – Représentation des comportements des agents loups et moutons

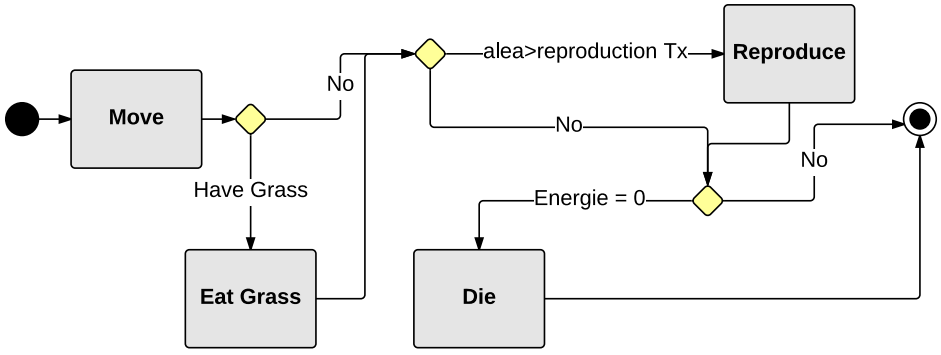


FIGURE 3.12 – Diagramme d'état transition des comportements d'un agent mouton

3.5 Distribution d'un modèle multi-agent avec les *Nested Graphs*

Dans cette partie nous nous intéressons à l'autre avantage de cette modélisation qui est, la distribution d'un modèle décrit par des *Nested Graphs* dans un environnement distribué en vue de son exécution parallèle. Pour illustrer cette distribution, nous utilisons à nouveau le modèle proie-prédateur. Après une description formelle de notre approche de distribution, nous détaillons notre proposition de manière schématique.

En utilisant les *Nested Graphs*, nous avons montré qu'un modèle peut être représenté sous la forme d'une structure basée sur des graphes imbriqués. Cette représentation nous permet de faire reposer la distribution du modèle sur des outils classiques de partitionnement distribué de graphes. L'équilibrage obtenu avec cette représentation est beaucoup plus fin que ce que nous avons pu faire avec les structures de grilles. Néanmoins, dans un modèle, tous les agents n'engendrent pas forcément la même charge de calcul, en particulier en fonction du type des agents. Il est donc nécessaire de compléter notre représentation à base de *Nested Graphs* pour prendre en compte cette information pour garantir un bon équilibrage.

Pour des raisons de clarté, uniquement les niveaux d'abstractions 0, 1 et 2 sont représentés sur les figures de cette section. Les comportements internes des agents définis par le niveau 3 d'abstraction ne sont pas représentés.

3.5.1 Description formelle

Dans notre proposition, la distribution est basée sur le poids assigné à chacun des agents ou plutôt chacun des *Agent Graphs* composant la simulation comme défini dans le chapitre 3 Equation 3.1. Formellement, la fonction de poids est définie par $\mathcal{W} : \tau \rightarrow \mathbb{N}$ et permet d'assigner à chaque agent un poids. Le poids assigné à chaque agent est défini par le modélisateur qui estime le poids à attribuer à chaque type d'agent. Par exemple, un type d'agent qui possède un grand nombre de comportements à réaliser ou alors des calculs couteux aura un poids plus élevé qu'un agent qui ne fait que de se déplacer sur un environnement. Grâce au poids assigné à chaque agent, nous pouvons effectuer la distribution en calculant la densité de chacun des nœuds en tenant compte des poids des agents qui sont contenus dans ces nœuds. La densité d'un *Agent Graph* $a = \langle \mathfrak{G}, \mathfrak{A}, \mathfrak{T}, \mathfrak{L}, \mathfrak{V} \rangle$ se définit par la fonction de densité suivante $\mathcal{W}^* : \mathbb{G} \mapsto \mathbb{N}$ comme :

$$\mathcal{W}^*(a) = \Sigma_{n \in \mathfrak{G}} \mathcal{W}^*(n) + \mathcal{W}(a) \tag{3.2}$$

Soit, $\Omega = \langle \mathfrak{G}, \mathfrak{A}, \mathfrak{T}, \mathfrak{L}, \mathfrak{V} \rangle$ la simulation à distribuer où, $\mathfrak{G}$ est un ensemble de cellules. Nous définissons le graphe de densité cette simulation Ω comme un graphe pondéré $\mathcal{D}_\Omega = \langle V, E, W_V, W_E \rangle$ où $V = \mathfrak{G}$ est un ensemble de nœuds, $E = \mathfrak{A}$ est un ensemble d'arcs orientés, $W_V : V \mapsto \mathbb{N}$ tel que $\forall\ a \in \mathfrak{G}, W_V(a) = \mathcal{W}^*(a)$ est une fonction qui assigne un poids aux nœuds à l'aide de la fonction de densité $\mathcal{W}^*$, et $W_E : V \mapsto \mathbb{N}$ tel que $\forall\ a \in \mathfrak{A}, W_E(a) = 1$ est une fonction qui assigne un poids à chacun des arcs. Il en résulte que la distribution de la simulation Ω sur k processus correspond à un problème de k-partitioning du graphe de densité $\mathcal{D}_\Omega$ où les nœuds de la i^{th} partition issue du découpage sont assignés au i^{th} processus. En effet, le problème de k-partitioning pour un graphe pondéré consiste à ce que les k partitions soit réparties de manière équilibrée en terme de poids. La charge de calcul de la simulation est donc efficacement répartie entre les k processus. De plus, comme le problème de k-partitioning d'un graphe pondéré minimise aussi la somme des poids des arcs coupés, alors la charge de communication associée à la répartition entre les k processus est également minimisée.

Nous illustrons cette proposition avec le modèle proie-prédateur. Ce modèle comporte trois niveaux d'abstraction (Figure 3.8). Le niveau d'abstraction le plus élevé représente la simulation dans son intégralité. Ce niveau est présent sur chacun des processus. Il est en effet nécessaire

pour supporter la distribution du modèle. Pour cette raison, il est considéré comme le niveau d'abstraction de base numéroté 0. Le second niveau d'abstraction est l'environnement (agent de type *Cell*) sur lequel la simulation se déroule. Comme énoncé dans le précédent chapitre les agents de type *Cell* sont un moyen de contenir les autres agents de la simulation qui évoluent sur l'environnement. L'environnement n'effectue pas de comportement, il n'engendre donc pas de charge. Pour cette raison, le poids qui lui est assigné est égal à 0. Les agents de type loup et mouton possèdent le même nombre de comportements définis dans la section précédente, à savoir se déplacer (Move), se nourrir (Eat), se reproduire (Reproduce) et mourir (Die). Ces agents engendrent donc la même charge lors de l'animation. Pour cette raison, le poids de chaque type d'agent (loup et mouton) est égal à 1. En assignant respectivement un poids à chacun des agents qui composent la simulation, il est possible de calculer la somme des poids au plus haut niveau d'abstraction. La somme des poids au plus haut niveau d'abstraction est égale la somme des poids de tous les agents des niveaux inférieurs. Dans le modèle proie-prédateur, le poids d'un agent de type *Cell* est donc égal à la somme des poids des agents qui le composent étant donné que le poids d'une cellule est égal à 0. De cette manière, il est possible de découper le graphe au plus haut niveau d'abstraction pour que chaque partie du graphe issue de ce découpage possède une somme de poids équivalente ou proche. Un exemple arbitraire de distribution du modèle proie-prédateur où le nombre de processus k est égal à 2 est représenté en Figure 3.13.

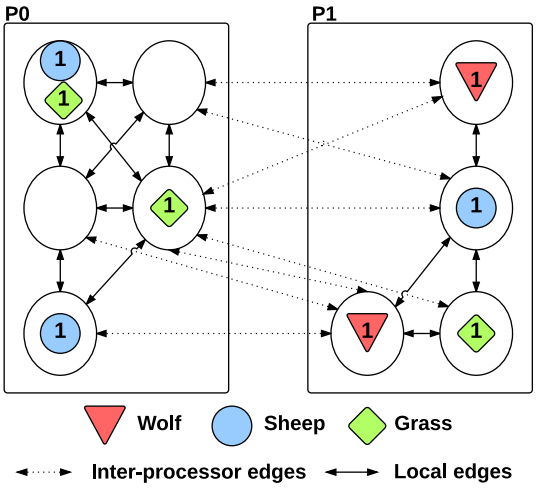


FIGURE 3.13 – Distribution du modèle proie-prédateur en utilisant les *Nested Graph* sur deux processus

3.5.2 Illustration

Après la description formelle de notre proposition de distribution, nous l'illustrons de manière graphique pour en faciliter la compréhension. La Figure 3.14 représente une configuration initiale arbitraire du modèle proie-prédateur basée sur les *Nested Graphs*.

La configuration initiale présentée sur la Figure 3.14 est composée de trois niveaux de *Nested Graph* correspondant aux différents niveaux présentés précédemment sur la Figure 3.8. Pour rappel, le premier niveau d'abstraction est l'environnement (niveau 0), il est représenté par une grande ellipse qui contient les niveaux inférieurs. Le second niveau d'abstraction représente l'environnement en forme de grille 3.9 (niveau 1). Pour finir le dernier et troisième niveau d'abstraction représente les agents loups et moutons (niveau 2). Cette configuration initiale représente donc notre simulation multi-agents Proie Prédateur. Notre objectif est de la distribuer sur plu-

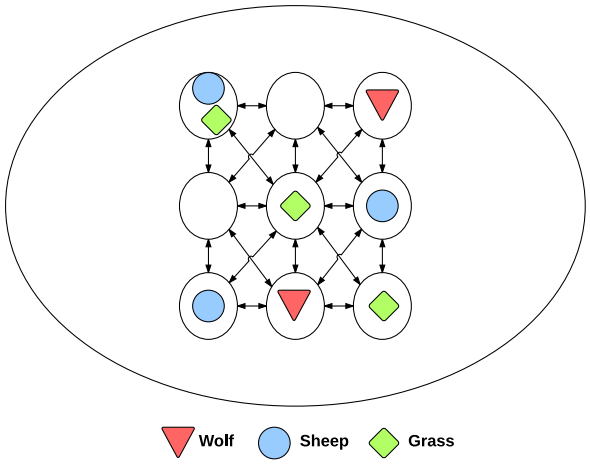
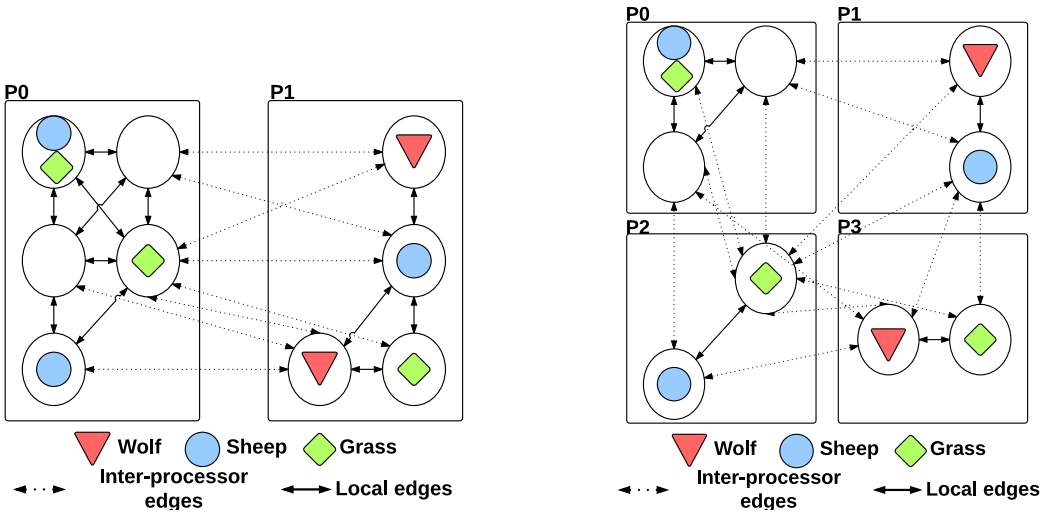


FIGURE 3.14 – Configuration initiale du modèle proie-prédateur avec les *Nested Graphs*

sieurs processus pour l'exécuter en parallèle. Pour utiliser de manière efficace des ressources parallèles, nous devons découper cette configuration ou structure initiale en plusieurs portions ou partitions qui seront ensuite assignées aux processus.

Avec la structure de *Nested Graph*, la distribution est calculée en fonction de la densité des agents contenus dans chacun des nœuds de niveau supérieur. Comme la structure de graphe n'est pas rigide, c'est à dire qu'elle offre un découpage plus souple alors la distribution est plus flexible. La Figure 3.15(a) présente la configuration initiale modélisé à l'aide des *Nested Graph* distribuée sur deux processus à l'aide du graphe de densité. Nous constatons que la répartition est plus équilibrée entre les processus et donc plus efficace qu'elle ne l'était avec une structure de grille. La charge est également mieux équilibrée lorsque que l'on distribue la simulation sur quatre processus comme le montre la Figure 3.15(b).



(a) sur deux processus

(b) sur quatre processus

FIGURE 3.15 – Exemples de distribution du modèle proie-prédateur à l'aide de la structure de *Nested Graphs*

Il faut noter que la distribution des agents de type *Cell* ainsi que l'équilibrage de charge sont fortement dépendant du modèle que nous simulons. Par exemple, dans le modèle proie-prédateur

la charge d'un agent mouton est identique à celle générée par un agent loup, en assignant un poids identique aux deux types d'agents, cela permet donc de réaliser une distribution bien équilibrée. Cependant, notre proposition permet également de distribuer des modèles où les agents ont des charges différentes permettant ainsi de retranscrire cette différence afin qu'elle soit prise en compte dans le calcul de la distribution. C'est la raison pour laquelle les poids assignés aux agents sont définis par le modélisateur de la simulation, comme paramètre d'entrée.

Du point de vue de la distribution, les *Nested Graphs* apportent deux niveaux de souplesse pour la mise en œuvre d'un système multi-agents sur des ressources distribuées. Le premier niveau est une indépendance par rapport à la structure de grille puisque les *Nested Graphs* ne traduisent que des liens entre entités et donnent donc la possibilité de s'affranchir de contraintes telles que les contraintes géographiques imposées par les grilles. Le second niveau est la possibilité d'assigner des pondérations aux entités pour tenir compte de la charge qu'elles engendrent. Cette possibilité s'avère nécessaire pour obtenir un équilibrage fin de la charge indispensable à l'efficacité de l'exécution d'une simulation.

3.6 Implémentation

Dans cette section, nous abordons dans un premier temps, les outils de partitionnement parallèles utilisés puis dans un deuxième temps, nous présentons la plateforme que nous avons développé appelée FractalPMAS. Cette plateforme repose sur le formalisme de *Nested Graphs* pour implémenter un modèle agent et l'animer.

3.6.1 Outils de distribution

Grâce à la structure de *Nested Graph*, nous pouvons tirer parti d'outils de partitionnement parallèles existants. Ces outils sont très performants et efficaces. Nous avons choisi la bibliothèque Zoltan [90] pour effectuer le partitionnement basé sur le graphe de densité d'agents. La bibliothèque Zoltan est une bibliothèque regroupant de nombreux algorithmes combinatoires pour les applications scientifiques parallèles. Cette bibliothèque possède en outre une suite d'algorithmes d'équilibrage de charge et de partitionnement dynamique qui augmentent la performance parallèle des applications en réduisant le temps d'inactivité des processus.

La bibliothèque Zoltan offre différents avantages, elle propose :

— des interfaces pour utiliser les bibliothèques Parmetis [82], PT-Scotch [83] sans avoir à réécrire les appels aux fonctions dans le code

— de bonnes performances pour le partitionnement de graphes à large échelle [91]

— une structure de données interne qui est complètement séparée de la structure de données de l'application sur laquelle le partitionnement est effectué. Cette séparation permet de ne pas contraindre les applications à utiliser une structure de données imposée par la bibliothèque. La séparation entre les structures (bibliothèque et application) s'effectue par l'intermédiaire de fonctions "callback" (e.g. `ZOLTAN_NUM_OBJ_FN`, `ZOLTAN_EDGE_LIST_MULTI_FN`. Les fonctions "callback" sont en réalité des fonctions écrites par l'utilisateur afin d'accéder à sa propre structure de données et donc de retourner à la bibliothèque les informations nécessaires à son bon fonctionnement. Des exemples de fonctions utilisées pour le partitionnement de graphes avec leur valeurs de retour sont décrits dans la Table 3.1.

Par exemple, en utilisant des fonctions "callback", il est aisé de calculer le nombre de nœuds qu'un processus possède ou encore le nombre d'arcs détenus par chaque nœud. Lorsque que l'application souhaite effectuer un partitionnement de graphe, elle fait appel à la bibliothèque Zoltan (e.g. Zoltan_LB_Partition) qui, à son tour, fait appel aux fonctions "callback" définies par l'utilisateur afin de pouvoir obtenir les informations qui lui permettent de réaliser le partitionnement.

Callback	Valeurs retournées
ZOLTAN_NUM_OBJ_FN	Nombre d'objets (ou agents) qu'un processus possède
ZOLTAN_OBJ_LIST_FN	Liste des identifiants et poids des objets (ou agents) qu'un processus possède
ZOLTAN_NUM_EDGE_MULTI_FN	Nombre d'arcs qu'un processus possède
ZOLTAN_EDGE_LIST_MULTI_FN	Liste des arcs et poids qu'un processus possède

TABLE 3.1 – Fonctions utilisées par la bibliothèque Zoltan et leurs valeurs de retour

L'intégration de la bibliothèque Zoltan dans une application s'effectue en cinq grandes étapes qui sont : l'initialisation, le partitionnement, la migration, la libération de la mémoire pour le partitionnement et la libération de la mémoire comme présenté dans la Figure 3.16.

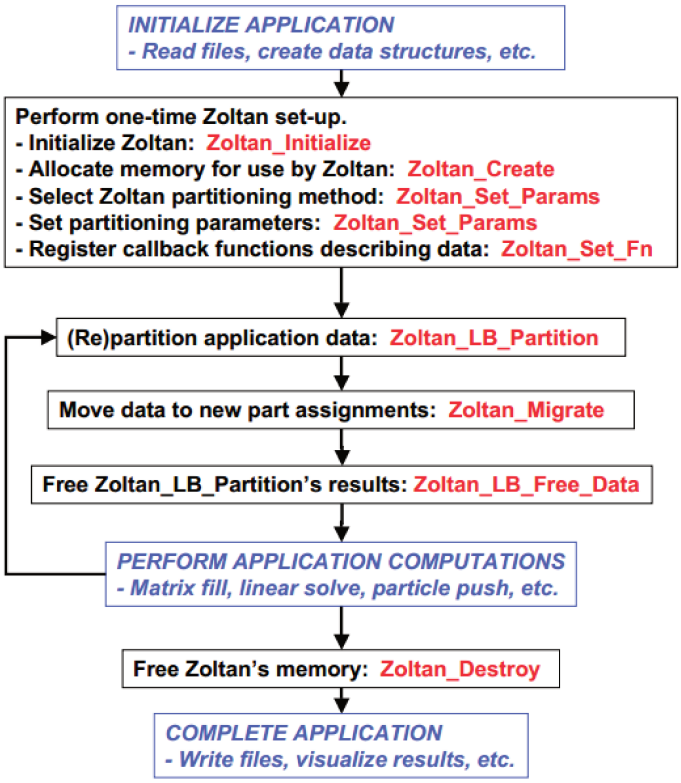


FIGURE 3.16 – Schéma d'utilisation de la bibliothèque Zoltan dans une application (Les appels à la bibliothèque sont préfixés par Zoltan [6]).

3.6.2 La plateforme FractalPMAS (FPMAS)

Afin de valider notre proposition, nous avons développé une plateforme multi-agents parallèle appelée FractalPMAS (FPMAS) qui repose sur le formalisme à base de *Nested Graph* pour implémenter un modèle agent ainsi que sur la bibliothèque Zoltan pour supporter la distribution de la simulation. La plateforme FPMAS est à l'étape de preuve de concept mais elle permet actuellement d'implémenter et d'exécuter des modèles agents avec le formalisme à base de *Nested*

Graph. Pour implémenter un modèle agent, nous utilisons la bibliothèque appelée CGraph [1] que nous avons développé en C++. La bibliothèque CGraph permet à l'aide de fonctions de créer des structures de *Nested Graph* pour représenter le modèle agent. Il est important de noter que dans cette plateforme, toute entité est considérée comme un agent. Chaque agent dans la plateforme est identifié grâce à un identifiant global unique. Une fois le modèle implémenté, FPMAS peut l'exécuter dans un environnement parallèle de type super-calculateur. Dans la plateforme, la simulation est divisée en n portions et chacune des portions est distribuée sur p processus qui ont à charge d'effectuer la simulation. Ces processus sont appelés *Agent Containers* (AC) et sont responsables de l'exécution des agents, de la réception des messages provenant des autres processus et de la délivrance des messages aux agents dont ils sont responsables. Les AC sont aussi responsables des migrations d'agents entre les processus. L'algorithme 1 décrit le fonctionnement global de la plateforme avec l'intégration de la bibliothèque Zoltan.

Algorithme 1 : Utilisation de la bibliothèque Zoltan pour réaliser le distribution de la simulation

```
1  ZoltanInitialisation();
2  ReadInitialFileData();
3  SynchronisationBarrier();
4  Zoltan_LB_Balance(...);
5  Zoltan_Migrate(...);
6  while (!End) do
7  |  Execute_One_Timestep();
8  |  Receive_Agents_Messages();
9  |  Deliver_Agents_Messages();
10 |  Zoltan_LB_Balance(...);
11 |  Zoltan_Migrate(...);
12 |  Synchronization_OLZ();
13 end while
```

L'étape d'initialisation (ligne 1) consiste à paramétrer la bibliothèque Zoltan en définissant les algorithmes à utiliser mais aussi à enregistrer les fonctions "callback" qui permettent le bon fonctionnement de la bibliothèque pour accéder à la structure de *Nested Graph*. Pour être plus efficace, la configuration initiale de la simulation est écrite dans un fichier qui est chargé au début de la simulation de manière parallèle (ligne 2) à l'aide des fonctionnalités de la bibliothèque MPI (MPI_File_open). Chaque processus, ouvre le fichier et lit la portion du fichier le concernant. La portion du fichier assignée à chaque processus est obtenue en divisant le nombre totale de lignes contenues dans le fichier par le nombre processus participant à l'exécution, ensuite en fonction du rang de chaque processus, il est aisé de calculé la ligne de début de chaque portion pour chaque processus. Après avoir lu le fichier et instancié le modèle dans la plateforme, les processus font appel à la bibliothèque Zoltan afin de vérifier si la distribution est correcte ou s'il est nécessaire de la mettre à jour (ligne 4). Cet appel va permettre de calculer une distribution du graphe. Les informations de sortie de l'étape de distribution seront alors les informations d'entrée de l'étape de migration (ligne 5). Durant l'étape de migration, la bibliothèque va déplacer des nœuds du graphe d'un processus à un autre pour respecter le calcul de la distribution. Dans la majorité des cas, il est nécessaire de mettre à jour la distribution car les agents de la simulation sont très rarement répartis de manière uniforme sur l'environnement. Une fois que la distribution est mise à jour et que la charge est correctement répartie alors nous pouvons exécuter les pas de temps

1. `https://github.com/LoW12/FractalGraph`

de la simulation jusqu'à une condition d'arrêt (ligne 6). A chaque pas de temps, les fonctions (lignes 8 et 9) réceptionnent et délivrent les messages de communication entre agents de la simulation. Ensuite, nous faisons de nouveaux appels aux fonctions de Zoltan pour équilibrer et migrer les agents si la charge n'est plus équilibrée (lignes 10 et 11). Pour finir, l'étape de synchronisation (ligne 12) permet de créer des zones de recouvrement, c'est à dire de copier les informations à proximité des zones de coupures pour garder la continuité de la simulation entre tous les processus participant à l'exécution.

Pour valider notre proposition nous effectuons des tests de performance dans la section suivante.

3.7 Expérimentations

Dans cette section, nous présentons des résultats de performance pour des d'exécutions utilisant la structure de *Nested Graph* afin de modéliser et de distribuer une simulation multi-agents. Pour effectuer les tests de performance, nous avons dans un premier temps implémenté le modèle proie-prédateur (défini au Chapitre 1 en section 1.1.5) dans notre plateforme FPMAS. Cependant, ce modèle n'est pas implémenté dans toutes les plateformes multi-agents parallèles à cause d'un manque de synchronisation pour les écritures distantes. Pour cette raison, nous avons aussi implémenté le modèle d'évaluation des SMA parallèles ou modèle de référence défini en section 1.1.5 afin de comparer les performances de la plateforme FractalPMAS avec celles des autres plateforme. Pour exécuter les simulations, nous utilisons la même configuration de cluster qui a été utilisée dans le Chapitre 2 à la section 2.3.1.

3.7.1 Réflexions sur la reproductibilité

Nous nous sommes efforcés de donner les paramètres d'exécutions des simulations pour être dans une démarche "Open Science". Mais suite aux expérimentations, on s'aperçoit qu'une variation existe d'une exécution à l'autre.

Nous définissons une simulation comme répétable si les résultats et le déroulement de la simulation sont exactement identiques d'une exécution parallèle à une autre.

En fait, le parallélisme ne permet pas de répéter exactement deux simulations parallèles à cause de la charge des nœuds, du réseau et des messages.

Il est important de noter que réaliser un PDMAS répétable exactement n'est pas atteignable à moins d'imposer une sérialisation des messages qui de fait rend caduque la parallélisation et impacte fortement les performances. Ce problème est évoqué pour la plateforme Flame [92] avec le modèle proie-prédateur. Pour pallier cet indéterminisme dans l'ordre des messages, la plateforme Flame propose des mécanismes de tri des messages permettant de garantir que chaque message sera toujours traité dans le même ordre d'une exécution à un autre, cependant cela est très couteux.

Notre objectif est donc plutôt d'atteindre des simulations reproductibles, c'est à dire d'obtenir des résultats à tendance identique d'une exécution à une autre.

Nous abordons le problème d'indéterminisme des messages dans le Chapitre 5 avec la communication entre agents.

3.7.2 Expérimentation avec le modèle proie-prédateur

Le modèle proie-prédateur utilisé pour les expérimentations est basé sur une grille de 1000x1000 qui représente l'environnement où 25000 moutons et 17000 loups sont initialement positionnés de manière aléatoire. Il est important de noter que pour des soucis de reproductibilité, la même

configuration initiale est utilisée pour toutes les exécutions des simulations, seule la graine varie d'une exécution à une autre. Le modèle utilisé pour les comportements et l'initialisation de l'énergie des agents loups et moutons est issu du modèle de la plateforme NetLogo [37]. L'énergie gagnée par un moutons lorsqu'il mange de l'herbe est fixée à 5 et à 20 lorsqu'un loup mange un mouton. En ce qui concerne les taux de reproduction ils sont fixés à 5 pour les moutons et à 4 pour les loups. Pour finir, les simulations sont exécutées durant 2000 pas de temps. La Table 3.2 présente les jeux de paramètres détaillés pour les expérimentations du modèle proie-prédateur.

Paramètres	Jeu de valeurs 1	Jeu de valeurs 2
Environnement	1000x1000	1000x1000
Nb. mouton	25000	25000
Nb. loup	17000	17000
Tx. reproduction mouton	0.5	0.5
Tx. reproduction loup	0.4	0.4
Gain nourriture mouton	4	4
Gain nourriture loup	20	20
Vie initiale mouton	4	4
Vie initiale loup	20	20
Croissance herbe	8	30
Nb. Pas de temps	2000	2000

TABLE 3.2 – Jeux de paramètres pour les expérimentations du modèle proie-prédateur

Les Figures 3.17 et 3.18 présentent des résultats d'une exécution de simulation du modèle proie-prédateur basé sur la structure de *Nested Graph* exécuté sur 128 cœurs. La Figure 3.17 représente un modèle avec de l'herbe qui croisse tous les 8 pas de temps tandis que la Figure 3.18 représente un modèle avec de l'herbe qui croisse tous les 30 pas de temps. Il est important de noter que deux exécutions ne donnent pas les même résultats et donc faire la moyenne sur plusieurs exécutions n'a pas de sens (une réfléxion sur la reproductibilité est effectué à la Section 3.7.1). Comme nous pouvons le constater, l'équilibre de l'écosystème est respecté même si la tendance d'une figure à une autre est différente. Cela valide le fait que notre approche obtient les mêmes résultats que les implémentations ou approches plus classiques.

Comme nous ciblons les machines parallèles, et plus particulièrement les super calculateurs, l'extensibilité de notre approche est importante. Cela permet de valider que notre approche reste viable même pour de grands modèles exécutés sur un grand nombre de processus. Les Figures 3.19 présentent les résultats d'extensibilité (Figure 3.19(a)) et de temps d'exécutions (Figure 3.19(b)) de notre approche pour le modèle proie-prédateur avec la plateforme FPMAS. Pour information, sur ces courbes, chaque point est la moyenne de 10 exécutions. Nous n'affichons pas le coefficient de variation sur le graphique car il est compris entre 0.1% et 0.6%, ce qui est relativement faible. Comme nous pouvons le constater la plateforme FPMAS passe bien à l'échelle jusqu'à 128 cœurs pour le modèle proie-prédateur et il n'y a pas de coupure dans la progression du speed-up qui est linéaire, même si ce dernier n'est pas optimal. Il est important de noter que le temps de référence utilisé pour calculer le speed-up est basé sur une exécution à 2 cœurs et ne peut donc pas être plus de la moitié du nombre de cœurs. Nous avons fait ce choix car la plateforme FPMAS ne peut pas être exécutée sur un seul cœur à cause de la synchronisation.

3.7.3 Expérimentation avec le modèle d'évaluation des SMA parallèles

Le modèle d'évaluation des SMA parallèles utilisé pour les expérimentations est basé sur une grille de 300x300 où 10000 agents sont initialement positionnés de manière aléatoire. De la même

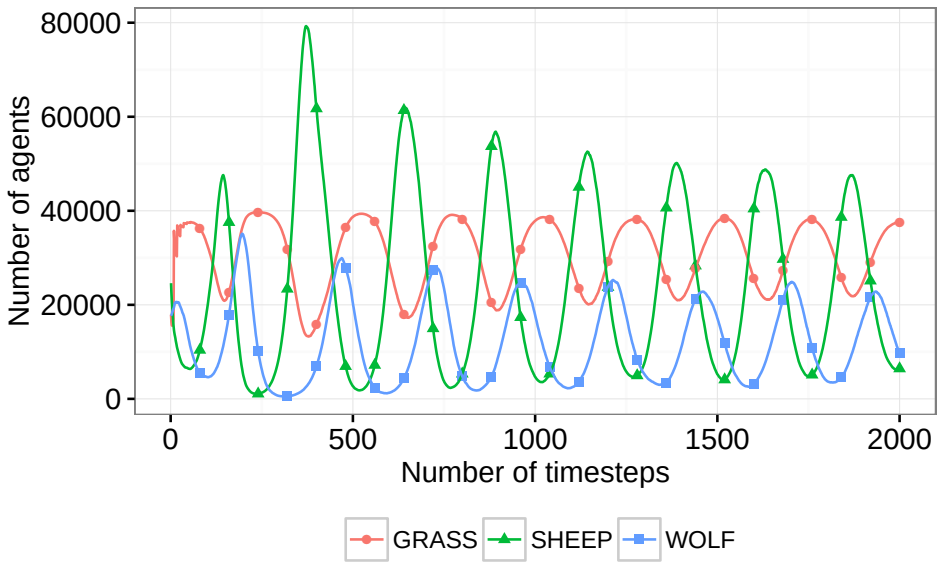


FIGURE 3.17 – Résultats d'exécution du modèle proie-prédateur utilisant les *Nested Graph* sur 128 cœurs et 2000 pas de temps (Jeu de valeurs 1 de la Table 3.2)

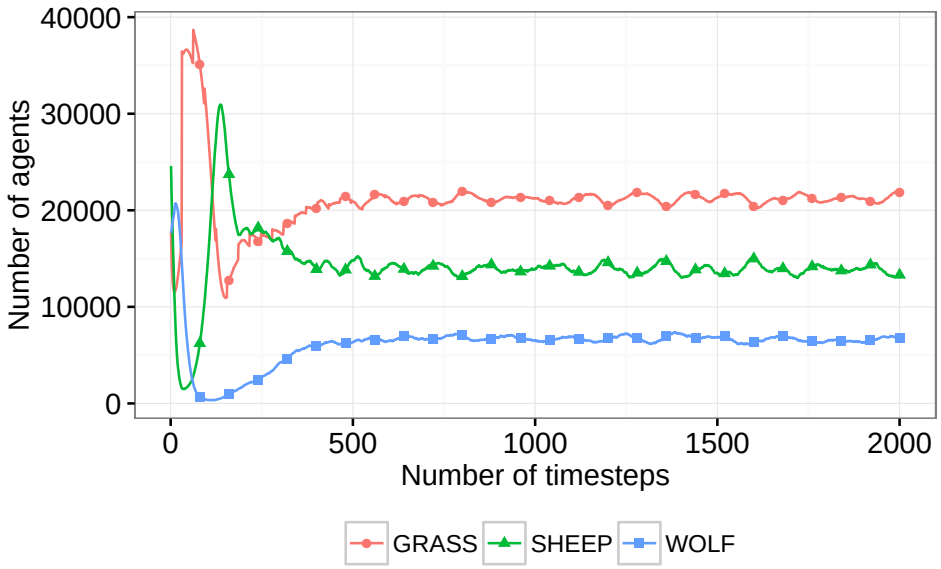
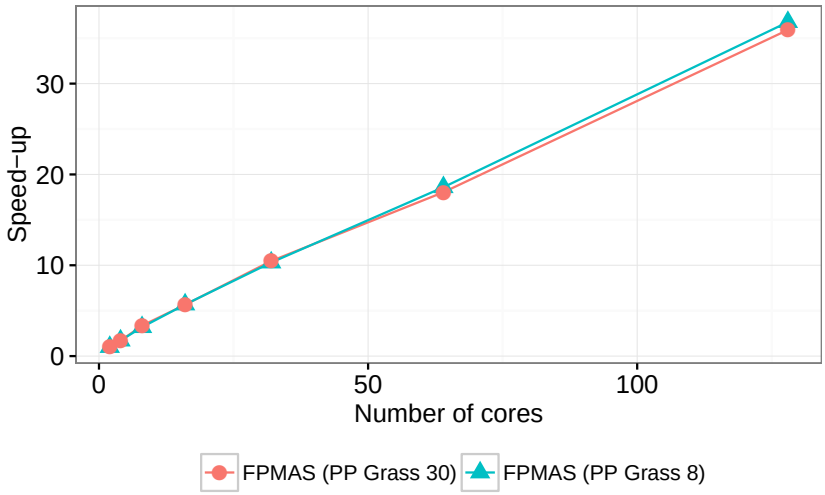
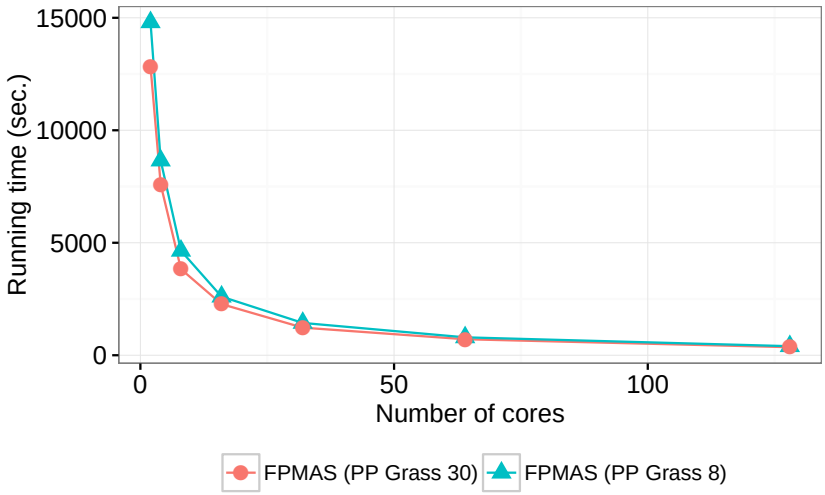


FIGURE 3.18 – Résultats d'exécution du modèle proie-prédateur utilisant les *Nested Graph* sur 128 cœurs et 2000 pas de temps (Jeu de valeurs 2 de la Table 3.2)

manière que pour le modèle proie-prédateur, par souci de reproductibilité, la même configuration initiale est utilisée pour toutes les exécutions des simulations et pour toutes les plateformes, seule la graine varie d'une exécution à une autre. A l'exception de la plateforme RepastHPC qui n'utilise pas de fichiers d'initialisation mais propose une méthode d'initialisation unique qui sera exécutée sur chacun des processus participant à la simulation. Le champ de perception des agents est de rayon 3. En ce qui concerne les choix aléatoires (choix de la cellule voisine pour le déplacement et l'initialisation de la table DFT), tous sont basés sur des lois uniformes. Le calcul



(a) Speed-up



(b) Running time

FIGURE 3.19 – Extensibilité et temps d'exécutions pour la plateforme FractalPMAS avec le modèle proie-prédateur basé sur la structure de *Nested Graph* de 2 à 128 cœurs pour 2000 pas de temps

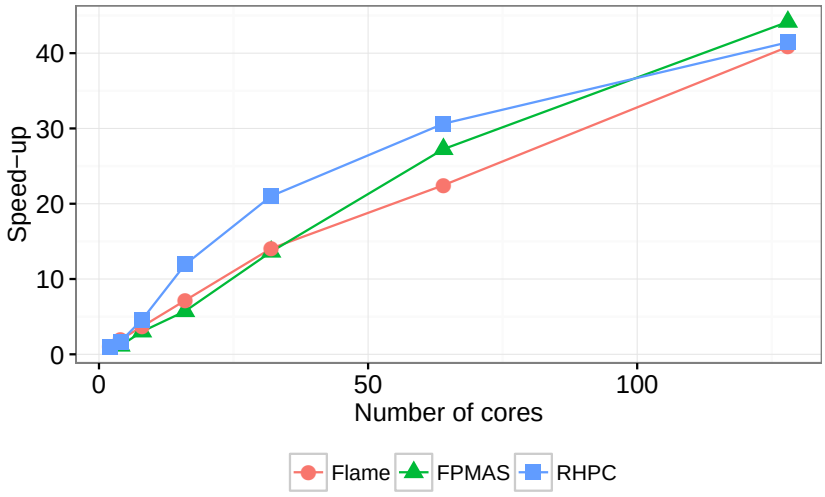
de DFT pour générer une charge interne à chaque agent est effectué sur un tableau de taille 128. Pour finir les simulations sont exécutés durant 200 pas de temps. La Table 5.1 présente les jeux de paramètres détaillés pour les expérimentations du modèle d'évaluation des SMA parallèles.

Sur les Figures 3.20 chaque point est une valeur moyenne de dix exécutions. Comme précédemment le coefficient de variation entre les résultats d'exécution est faible, compris entre 0,1% et 0,7%. Nous ne l'avons donc pas reporté sur les graphiques.

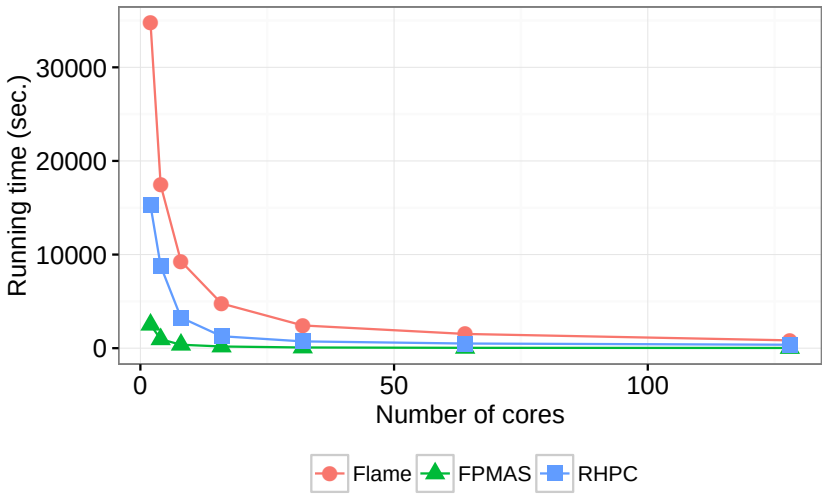
Les Figures 3.20 montre l'extensibilité (Figure 3.20(a)) et les temps d'exécutions (Figure 3.20(b)) des plateformes avec l'exécution du modèle de référence. Comme pour le précédent modèle proie-prédateur, le temps de référence pour calculer le speed-up est basé sur 2 cœurs. A partir de ces figures, nous pouvons conclure que les trois plateformes sont extensibles jusqu'à 32 cœurs même si RepastHPC à tendance à croître plus lentement que les autre plateformes après 64 cœurs.

Paramètres	Jeu de valeurs 1
Environnement	300x300
Nb. Agent	10000
Chp. perception	3
Taille DFT	128
Nb. pas de temps	200

TABLE 3.3 – Jeux de paramètres pour les expérimentations du modèle de référence



(a) Speed-up



(b) Running time

FIGURE 3.20 – Extensibilité et temps d'exécutions pour les plateformes FractalPMAS, RepastHPC et FLAME avec le modèle de référence de 2 à 128 cœurs pour 200 pas de temps

En ce qui concerne FPMAS elle est au dessus des autres plateformes pour des exécutions avec plus 100 cœurs. Nous obtenons cependant, un meilleur speed-up avec ce modèle qu'avec le proie-prédateur. Cela nous permet de valider notre approche basé sur les *Nested Graph* pour modéliser

les simulations multi-agents.

Comme la plateforme FPMAS n'est qu'à l'état de preuve de concept, il reste probablement encore beaucoup d'optimisations à effectuer. Cependant, il est encourageant de noter que FPMAS surpasse les autres plateformes lors de l'utilisation de plus de 100 cœurs. Cela signifie que notre approche de *Nested Graph* est adaptée pour les implémentations parallèles de simulations multi-agents.

Après avoir expliqué les différents principes utilisés pour modéliser, distribuer et exécuter notre simulation sur plusieurs processus, nous abordons les problèmes de cohérences de données, ou plutôt d'accès concurrents inhérents aux systèmes parallèles, dans le chapitre suivant où nous détaillons l'impact de la synchronisation sur l'exécution des modèles multi-agent parallèle.

Chapitre 4

Les pistes de synchronisation et leurs impacts sur les modèles

Sommaire

Pour simuler plusieurs centaines de milliers d'agents, et donc des millions d'interactions entre agents, il est nécessaire de distribuer la simulation multi-agents afin de l'exécuter sur plusieurs processus. L'une des manières pour l'effectuer est d'utiliser une politique de synchronisation forte (*Strong Synchronisation*). Mais cette synchronisation implique un coût en communications pour maintenir les processus dans état cohérent qui a un impact direct sur les temps d'exécution des simulations. Cette synchronisation forte permet d'éviter les problèmes de cohérence de données ou d'accès concurrents inhérents aux systèmes parallèles (HPC), les systèmes multi-agents parallèles sont donc soumis aux mêmes règles.

La synchronisation semble être un des points clefs pour une exécution efficace d'une simulation multi-agents parallèle du fait des nombreux échanges et dépendances temporelles qu'elle induit. Notre objectif dans ce chapitre est donc d'étudier l'impact de la synchronisation sur les temps d'exécution, mais aussi l'impact sur les résultats de simulations si nous relaxons la synchronisation dans les simulations. En effet, la synchronisation mise en place lors de l'implémentation dépend du modèle lui-même. Par exemple, si nous implémentons un modèle où les agents n'ont pas d'interaction, directe ou indirecte (à travers l'environnement), alors aucune synchronisation n'est nécessaire. D'autre part, les systèmes multi-agents permettent l'émergence de comportements globaux basés sur les interactions entre les entités qui composent le modèle. Par conséquent, l'observation d'un phénomène s'effectue au niveau macroscopique et non microscopique. Étant donné que de grandes quantités d'agents sont simulées, on peut supposer que cette quantité d'agents compense les synchronisations erronées ou fausses.

Premièrement, le problème est de savoir si la quantité d'agents permet de compenser le manque de synchronisation en offrant des résultats avec une tendance macroscopique identique. Deuxièmement, nous cherchons à quantifier le surcoût du à la synchronisation.

Dans la suite de ce chapitre, nous abordons dans un premier temps, les points de synchronisation nécessaires pour le bon déroulement d'une simulation agents parallèles, puis nous définissons plusieurs politiques de synchronisation afin d'évaluer leur impact sur les résultats et le temps d'exécution à l'aide d'expérimentations.

4.1 Quand synchroniser ?

La synchronisation dans une simulation distribuée a pour objectif de garantir l'accès à des données à jour mais aussi de garantir la cohérence des actions qui sont effectuées, c'est à dire que les règles du modèle à simuler soit respectées. Par exemple, dans le cas du modèle proie-prédateur, deux loups, ne peuvent pas manger le même mouton. Ceci est une violation des règles du modèle.

De manière générale il existe deux modes de fonctionnement de simulations : le mode *Ghost* et le mode non *Ghost* comme précédemment expliqué dans le Chapitre 1 à la section 1.2.2. Pour rappel, le mode *Ghost* consiste à utiliser les données calculées auparavant, au pas de temps ou à l'itération précédente, pour réaliser le calcul. Les résultats des calculs sont, eux, mémorisés dans une copie de travail. De ce fait, les processus sont obligés de conserver à la fois ces données calculées auparavant, le *Ghost*, et les résultats des calculs, la copie de travail. De cette manière, les informations du *Ghost* ne sont accessibles qu'en lecture et ne sont pas les dernières valeurs calculées. Dans le cadre des systèmes multi-agents, l'utilisation d'un modèle *Ghost* est possible uniquement avec certains modèles. Les modèles concernés sont ceux ne nécessitant pas d'écriture. En effet, utiliser le mode *Ghost* avec des écritures n'a que peu de sens car cela conduirait à la situation énoncée précédemment pour le modèle proie-prédateur : deux loups pourraient manger la même proie. En contraste, le mode non *Ghost* consiste à ne disposer que d'une seule instance des données, dans laquelle les résultats sont directement reportés. De ce fait certains calculs sont réalisés directement avec la version des données calculée lors du pas de temps ou de l'itération courante. Avec ce mode, les informations sont accessibles en lecture et en écriture mais nécessitent des mécanismes de synchronisation pour les données détenues par d'autres processus. Pour les simulations multi-agents parallèles, plusieurs points de synchronisation sont donc nécessaires au bon déroulement de la simulation qui sont les suivants :

La fin d'un pas de temps : lorsque l'on exécute la simulation en parallèle, chaque pas de temps est exécuté par tous les processus qui participent à la simulation. A la fin de chaque pas de temps une étape de synchronisation est nécessaire pour garantir que chaque processus exécute le même pas de temps et le même point de synchronisation.

La migration d'un agent : lorsqu'un agent doit se déplacer d'un processus à un autre pour continuer à exécuter ses comportements, une synchronisation est nécessaire pour migrer l'agent. En effet, le fait de transférer des données d'un processus à un autre implique que chaque processus doit exécuter le même pas de temps. De ce fait, une synchronisation dans la partie où l'on effectue la migration des données est nécessaire pour garantir que les processus concernés sont dans le même état.

La mise à jour des zones de recouvrement : pour garder la continuité des champs de perception des agents lors de la distribution de l'environnement sur plusieurs processus, un point de synchronisation est obligatoire pour mettre à jour les zones de recouvrement. Ces zones sont composées d'une copie partielle des cellules voisines. Lors de la mise à jour de ces zones de recouvrement, la synchronisation permet de garantir, de la même manière que pour la migration, que les différents processus concernés exécutent bien le même pas de temps et le même point de

synchronisation.

Ces trois points de synchronisation, sont nécessaires au bon déroulement d'une simulation agent parallèle qu'elle soit en mode *Ghost* ou non *Ghost*. Cependant, pour le mode non *Ghost*, ces points de synchronisation ne sont pas suffisants car ils ne permettent pas de gérer les écritures dans les zones de recouvrement. En effet, lors de l'exécution d'une simulation en mode *Ghost*, les actions du pas de temps n sont effectuées sur les données obtenues au pas de temps $n-1$ c'est à dire au pas de temps précédent, les problèmes d'écritures ne sont donc pas à gérer. En revanche, dans une simulation en mode non *Ghost*, les calculs sont effectués sur les données courantes, c'est à dire au pas de temps n. L'accès aux données doit être géré pour garantir qu'aucune incohérence (ou biais) n'est injectée dans la simulation et que les informations sont à jour. Dans les modèles agents, il existe des modèles qui nécessitent uniquement de la lecture, d'autres de la lecture et de l'écriture. Nous analysons les différents types de modèles dans la section suivante.

4.2 Analyse des modèles

Pour évaluer l'impact de la synchronisation sur les résultats d'exécution des simulations multi-agents, nous utilisons les trois modèles agents (Proie-prédateur, Virus et Flocking) définis au Chapitre 1 à la section 1.1.5. Ces modèles sont choisis car ils nécessitent plus ou moins de support de synchronisation pour s'exécuter sans incohérence. En effet, chaque modèle est soumis à ses propres contraintes de synchronisation, en particulier, au niveau de la gestion des écritures, concurrentes ou non.

Il est important de noter que le système modélisé et son implémentation ont un impact important sur la synchronisation qui doit être mise en place pour garantir la qualité des résultats obtenus par la simulation parallèle. Ceci est, en particulier, vrai pour le choix d'implémentation en utilisant un *Ghost* de l'état du système ou non. Ainsi, parmi les modèles suivants, nous avons choisi l'approche que nous avons le plus souvent trouvée par rapport à ce choix d'implémentation. Changer ce choix modifierait les contraintes de synchronisation et conduirait à d'autres conclusions.

Le modèle Flocking

Le modèle Flocking est un modèle qui ne nécessite aucune gestion d'écriture concurrente. Les agents oiseaux calculent leur déplacement en fonction de la position des oiseaux voisins obtenue au pas de temps $n-1$, c'est à dire au pas de temps précédent. Ce modèle fonctionne donc en mode *Ghost*. Les données des autres agents ne sont accédées qu'en lecture sur les données *Ghost*. De ce fait la mise à jour des données des zones de recouvrement à chaque pas de temps permet de garantir à chaque agent oiseau qu'il dispose des informations correctes pour calculer son déplacement. De même il n'y a pas de problème de concurrence d'accès sur ces données puisqu'elle ne sont accédées qu'en lecture seule.

Le modèle Virus

Le modèle Virus fonctionne, quant à lui, en mode non *Ghost*, il nécessite de ce fait une gestion de l'écriture concurrente. En effet, chaque agent contaminé peut infecter les agents présents dans son voisinage avec le virus. L'infection des agents s'effectue par l'appel d'une méthode sur les agents voisins. L'exécution étant répartie sur plusieurs processus, plusieurs agents exécutés sur plusieurs processus peuvent infecter simultanément un même agent. Puisque le modèle n'utilise pas de données *Ghost* il est possible que, dans un même pas de temps, un agent A, qui a été infecté, infecte à son tour un agent B. Si ceci arrive alors que les deux agents ne sont pas sur le

même site alors il est nécessaire de mettre à jour des données distantes. Puisque c'est l'agent A qui infecte, il doit appeler une méthode sur le processus de l'agent B. Or cette méthode doit être exécutée au plus tôt puisque l'agent B n'a peut-être pas encore été exécuté dans son processus et qu'il devra s'exécuter en tant qu'agent infecté. A noter que nous sommes confrontés ici à la difficulté de garantir la mise à jour immédiate si on veut que le système parallèle reproduise fidèlement le système centralisé. Pour finir, nous pouvons remarquer qu'un agent ne peut être infecté qu'une seule fois. En effet, une fois que l'agent est infecté, il ne change pas son état si il est contaminé par un autre agent. Cette propriété fait que nous n'avons pas à gérer de concurrence en écriture sur le changement de l'état de l'agent, puisque même si deux agents infectent un même agent l'ordre des deux exécutions n'a pas d'incidence sur le résultat final.

Le modèle proie-prédateur

Le modèle proie-prédateur fonctionne lui aussi sans conserver de données *Ghost*. En effet, dans la mesure où les prédateurs mangent les proies, ils en changent l'état, ce qui engendre une écriture (le changement de l'état vivant à celui de mort) dans les données de l'agent. Comme pour le modèle virus cela oblige à mettre en place une synchronisation pendant le pas de temps pour prendre en compte la modification.

Le modèle proie-prédateur est plus sensible que le modèle Virus aux synchronisations car il nécessite en plus de gérer les écritures concurrentes. Si une proie se situe dans une zone de recouvrement, à cause du parallélisme, des prédateurs situés dans des processus différents peuvent être tentés de manger une même proie puisque celle-ci sera présente à la fois dans la partie initiale de l'environnement et dans chacune des zones de recouvrement. Si tous les prédateurs mangent cette proie (celle qui est dans la zone de recouvrement) alors chacun bénéficiera d'un apport en énergie et donc d'une augmentation de sa durée de vie, ce qui constitue une erreur par rapport au modèle séquentiel. Il est donc nécessaire de synchroniser tous les agents qui souhaiteraient manger une proie pour garantir qu'un seul prédateur la mangera. De plus, pour rester cohérent avec le modèle séquentiel, il est nécessaire que la proie disparaisse des zones de recouvrement dès qu'elle est mangée puisque d'autres agents pourraient, dans la suite de l'exécution du pas de temps, être également tentés de la manger alors qu'elle n'est plus là.

L'analyse de ces modèles met en évidence différents besoins en terme de synchronisation en fonction des modèles. Ces besoins sont dictés par la modélisation. Ils dépendent donc des choix réalisés par le modélisateur et il n'est pas possible d'intervenir au niveau de ces choix une fois le modèle finalisé. Par contre, il est possible d'assister le modélisateur à la fois avec des recommandations sur l'implémentation et avec des outils de synchronisation. Par exemple, si le modélisateur pense que le choix d'une implémentation ou d'une autre, n'impacte pas la validité des résultats de simulation alors autant lui permettre de faire le choix de l'implémentation qui limite les problèmes de synchronisation. Pour assister le modélisateur et pour qu'il puisse évaluer l'impact de la synchronisation, nous définissons dans la section suivante plusieurs politiques de synchronisation.

4.3 Les politiques de synchronisation

Plusieurs politiques de synchronisation sont définies afin de pouvoir évaluer l'impact de la synchronisation sur les résultats mais aussi sur le temps d'exécution. Ces politiques sont définies à partir des constatations faites sur les modèles de base et parce qu'elles nous ont parues pertinentes sur la base des modèles que nous avons étudiés. Il est possible que d'autres politiques puissent présenter un intérêt dans le cadre de modèles spécifiques. A ces trois politiques

nous avons associé un cas sans synchronisation qui doit servir de référence. Les politiques de synchronisations définies sont :

Aucune synchronisation (NS) : Ce niveau consiste à distribuer la simulation en n portions en autorisant les agents à se déplacer d'un processus à un autre en ayant un champ de perception tronqué lorsqu'ils sont proches des limites des cellules. ce niveau comporte uniquement la migration des agents, aucune zone de recouvrement ou écriture distante n'est gérée.

Synchronisation par zones de recouvrement (OverLapping Zones - OLZ) : c'est le niveau de synchronisation le plus faible. Il consiste à mettre en œuvre uniquement les zones de recouvrement, c'est à dire la copie partielle des cellules voisines à chaque pas de temps. Aucune écriture dans les zones de recouvrement n'est gérée, c'est à dire que toutes les écritures dans les zones de recouvrement sont écrasées par la mise jour, issue du processus qui possède cette zone, à la fin du pas de temps et que les informations ne sont jamais reportées sur l'agent original. Cela signifie que le modèle, comme le modèle Flocking, peut bénéficier des données contenues dans la zone mais ne peut pas les modifier.

Écritures asynchrones (AS) : Ce niveau de synchronisation permet de faire des écritures à distance sans attendre une confirmation ou une valeur de retour suite à cette écriture. Il s'agit donc bien d'une écriture asynchrone. Ce niveau est utilisé lorsqu'un agent souhaite modifier une donnée dans la zone de recouvrement, que cette écriture doit avoir lieu au plus tôt pour être prise en compte dans le pas de temps courant, sans que l'agent qui réalise cette écriture attende une donnée en retour. Ce niveau est illustré par le cas du modèle virus où un agent peut contaminer un autre agent proche sans que le fait que l'agent cible soit contaminé ou non n'ait d'incidence pour l'agent contaminant.

Forte synchronisation (SS) : c'est le niveau de synchronisation le plus fort. Il consiste à mettre en œuvre les zones de recouvrements puis à gérer les écritures concurrentes pour garantir au maximum la reproduction du cas séquentiel. Dans ce niveau de synchronisation, chaque demande en écriture est bloquante jusqu'à l'obtention d'une réponse, ainsi la cohérence des données est garantie. C'est le niveau de synchronisation dont nous avons besoin pour implémenter le modèle proie-prédateur. Ce niveau représente ce que nous pouvons faire de mieux en synchronisation sans pour autant imposer une exécution séquentielle.

Forte synchronisation décalée (SSD) : ce niveau est identique à la forte synchronisation (SS) à la différence que les demandes d'écriture d'un agent ne bloquent pas les agents suivants dans l'ordre d'exécution. Ainsi, un agent qui fait une demande en écriture est suspendu ou mis en attente pour être géré à la fin du pas de temps.

Pour diriger le choix d'une politique de synchronisation pour un modèle, il est nécessaire de connaître l'incidence qu'aura le choix de cette politique sur le modèle parallélisé. Les spécifications données précédemment permettent de discerner l'impact sémantique. A partir des propriétés de chacun des niveaux de synchronisation, il est possible de savoir quelles contraintes seront garanties. Dans le cas, où le choix sur l'une ou l'autre politique est indécis, l'incidence de la politique de synchronisation sur les performances de la simulation s'avère utile. Nous présentons donc, dans la prochaine section, les mesures de performance qui ont été réalisées avec ces politiques.

4.4 Expérimentations

Dans cette section, nous présentons les résultats de performances obtenus lors de l'exécution des différents modèles avec les différentes politiques de synchronisation.

Pour exécuter les simulations, nous avons utilisé la même configuration de cluster que dans le Chapitre 2 à la section 2.3.1. Chaque point des courbes représente une moyenne de 10 exécutions avec 10 graines différentes. Dans un premier temps, nous présentons les résultats d'extensibilité des modèles avec les différentes politiques de synchronisation. Ensuite, nous analysons l'impact de la relaxation de la synchronisation sur les résultats issus des simulations. Pour finir, nous terminons ce chapitre par une discussion autour de la relaxation de la synchronisation dans les systèmes multi-agents parallèles.

4.4.1 Analyse de l'extensibilité

L'extensibilité des modèles est réalisée en fixant le nombre d'agents de la simulation et en faisant varier le nombre de processus sur lesquels la simulation s'exécute.

Modèle Flocking

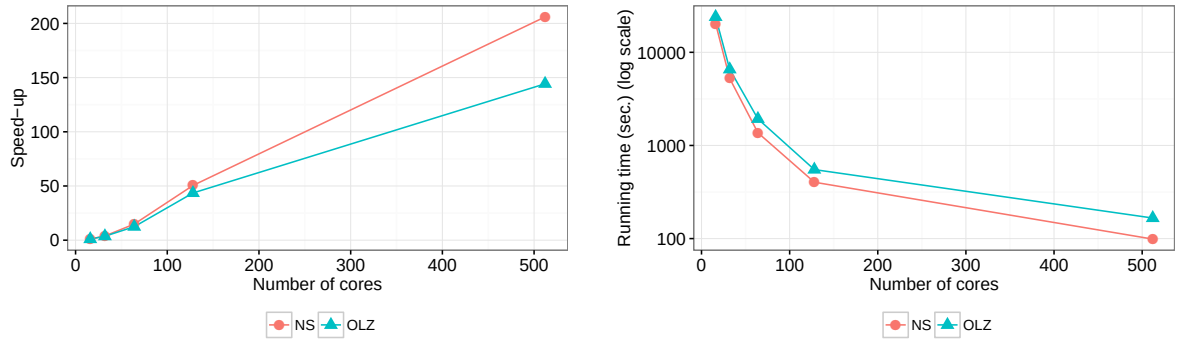
Le modèle Flocking utilisé pour les expérimentations est basé sur un cube de 1000x1000x1000 qui représente l'environnement où les oiseaux peuvent évoluer. A l'initialisation, 100000 oiseaux sont répartis de manière aléatoire dans l'espace. Il est important de noter que pour des soucis de reproductibilité, la même configuration initiale est utilisée pour toutes les exécutions des simulations, seule la graine varie d'une exécution à une autre. Les taux de cohésion, de séparation et d'alignement sont fixés à 1 tandis que le taux d'aléa est lui fixé à 1.5. Ces paramètres ont été choisis, afin de ne pas favoriser l'un des trois critères composant les comportements des oiseaux. Seul, le taux d'aléatoire est supérieur aux autres taux, dans le but de générer des déplacements plus chaotique et donc de tester davantage la synchronisation. Pour finir chaque simulation est exécutée durant 2000 pas de temps. La Table 4.1 présente les jeux de paramètres détaillés pour les expérimentations du modèle Flocking.

Paramètres	Jeu de valeurs 1
Environnement	1000x1000x1000
Nb. oiseau	100000
Tx. cohésion	1
Tx. séparation	1
Tx. aléatoire	1.5
Tx. alignement	1
Nb. Pas de temps	2000

TABLE 4.1 – Jeux de paramètres pour les expérimentations du modèle Flocking

La Figure 4.1 présente l'extensibilité 4.1(a) et les temps d'exécution 4.1(b) du modèle Flocking avec le jeu de valeurs 1 de la Table 4.1 pour les deux politiques de synchronisation *NS* et *OLZ*. Le temps de référence pour calculer l'extensibilité est basé sur 16 cœurs car une simulation avec 100000 d'agents ne s'exécute pas sur 8 cœurs pour des raisons de mémoire. Les valeurs de speed-up obtenues s'expliquent par le fait que lors de la découverte du voisinage d'un agent, une recherche non optimisée est effectué. En effet, une recherche dans tout l'environnement est effectuée, le nombre d'agents présents dans la surface est donc carré ce qui implique que lorsque le nombre de cœurs augmentent cela diminue aussi de manière carrée. A noter qu'il n'est pas

nécessaire d'évaluer les deux politiques SS et SSD sur ce modèle puisqu'ils n'engendrent pas de modifications dans les zones de recouvrement.



(a) Speed-up

(b) Running time

FIGURE 4.1 – Extensibilité et temps d'exécution du modèle Flocking (Jeu de valeurs 1 de la Table 4.1)

Comme nous pouvons le constater sur la Figure 4.1(a), l'extensibilité des deux courbes, avec et sans synchronisation, est bonne. Sans surprise, la courbe sans synchronisation offre de meilleurs résultats.

Sur la figure 4.1(b), pour 16 cœurs la différence des temps d'exécution est tout de même d'environ 15% , pour 128 cœurs la différence est d'environ 26%, et pour 512 cœurs la différence est de 65% , même si les courbes semblent très proches. Cet accroissement avec le nombre de cœurs s'explique simplement par le fait que plus de messages sont nécessaires pour mettre à jour les zones de recouvrement.

Modèle Virus

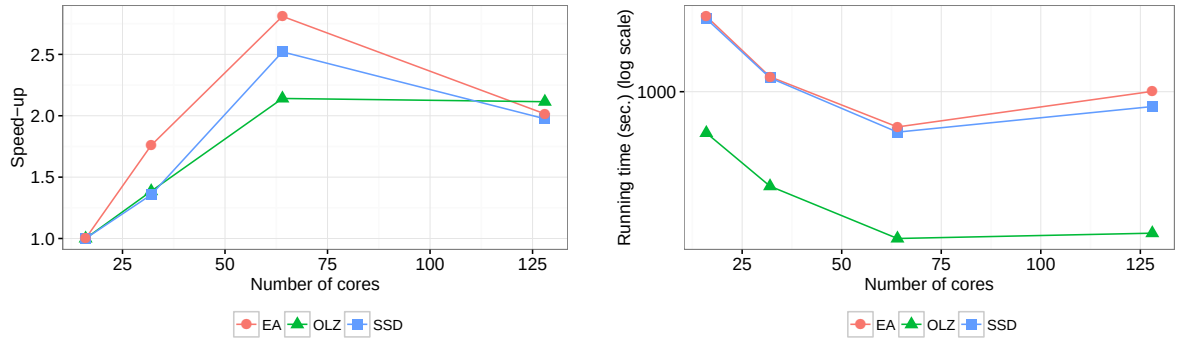
Le modèle Virus utilisé pour les expérimentations est basé soit sur une grille de 300x300 ou une grille de 1000x1000 qui représente l'environnement où la capacité maximale est respectivement de 800000 et 500000 personnes. A l'initialisation 9600 personnes sont saines et 640 sont infectées par le virus. Tous les agents sont positionnés de manière aléatoire sur l'environnement. Il est important de noter que, pour des soucis de reproductibilité, la même configuration initiale est utilisée pour toutes les exécutions des simulations, seule la graine varie d'une exécution à une autre. Le taux d'infection est fixé à 0.65 et le taux de reproduction est fixé à 0.2. En ce qui concerne le taux de récupération, c'est à dire, le fait qu'une personne infectée devienne immunisée, il est fixé à 0.5. Ces valeurs sont issues du modèle Virus [35], implémenté dans la plateforme NetLogo [25]. Seule la taille de l'environnement et la capacité maximale ont été adaptées pour obtenir un modèle de grande taille. Pour finir, chaque simulation est exécutée durant 800 pas de temps. La Table 4.2 présente les jeux de paramètres détaillés pour les expérimentations du modèle Virus. Le temps de référence pour calculer l'extensibilité est basé sur 16 cœurs.

La Figure 4.2 présente l'extensibilité (4.2(a)) et les temps d'exécution (4.2(b)) du modèle Virus avec le jeu de valeurs 1 de la Table 4.2 pour les trois politiques de synchronisation *OLZ*, *SSD* et *EA*.

Comme nous pouvons le constater sur la Figure 4.2(a), l'extensibilité décroît à partir de 64 cœurs. Cette décroissance de l'extensibilité s'explique par le fait que la taille du modèle est trop petite pour pouvoir être parallélisée sur un nombre plus important de processus. En

Paramètres	Jeu de valeurs 1	Jeu de valeurs 2
Environnement	300x300	1000x1000
Capacité env.	800000	500000
Nb. person. init.	9600	9600
Nb. person. infec. init	640	640
Tx. infection	0.65	0.65
Tx. récupération	0.5	0.5
Tx. reproduction	0.2	0.2
Nb. Pas de temps	800	800

TABLE 4.2 – Jeux de paramètres pour les expérimentations du modèle Virus



(a) Speed-up

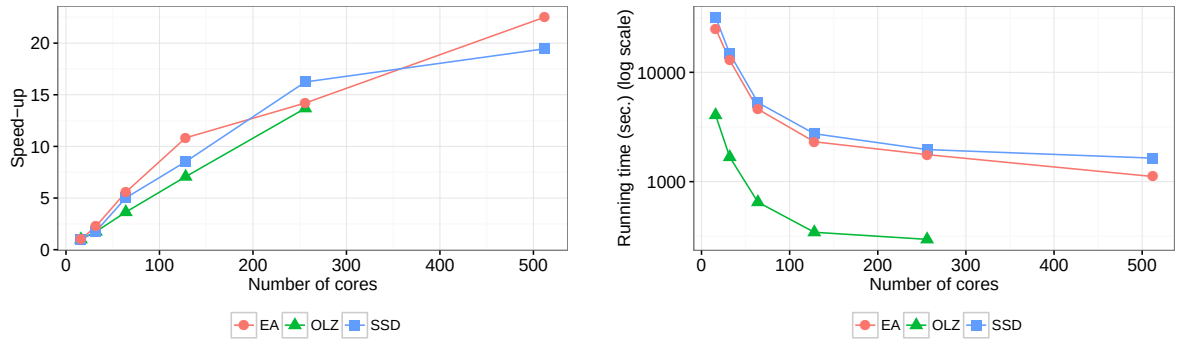
(b) Running time

FIGURE 4.2 – Extensibilité et temps d'exécution du modèle Virus (Paramètre 1 de la Table 4.2)

effet, le volume et le coût des communications est supérieur à celui des calculs à réaliser. La Figure 4.2(a) permet néanmoins de mettre en évidence le fait que les modèles ont une limite dans l'extensibilité de leur parallélisation ce qui nous conduit à avoir une borne pour le temps minimal d'exécution. Pour cette raison, la Figure 4.3 présente elle aussi l'extensibilité (4.3(a)) et les temps d'exécution (4.2(b)) du modèle Virus mais avec le jeu de valeurs 2 de la Table 4.2 pour les trois politiques de synchronisation *OLZ*, *SSD* et *EA*.

Comme nous le constatons sur la Figure 4.3(a), les trois courbes sont extensibles jusqu'à 512 cœurs. Nous disposons maintenant d'un modèle d'une taille suffisante. Aucun calcul pour la valeur de 512 cœurs n'est présent pour la courbe OLZ sur la Figure 4.3(a) car nous n'avons pas eu assez de temps et de disponibilité de ressources sur le calculateur. On remarque que la courbe utilisant l'écriture asynchrone (EA) offre de meilleurs résultats par rapport à la courbe utilisant la synchronisation forte décalée (SSD). Les résultats se confirment avec la Figure 4.3(b). En effet, pour 16 cœurs la différence au niveau des temps d'exécution des deux courbes (SSD et EA) est d'environ 27%, pour 128 cœurs la différence est seulement de 15% tandis que pour 512 cœurs la différence est de 47%. Nous expliquons ce surcout par deux raisons, premièrement par le traitement en fin de pas de temps des synchronisations, et deuxièmement par l'augmentation du volume de messages nécessaires à la synchronisation. Ce qui se confirme par le fait que la différence devient plus grande en augmentant le nombre de cœurs. En effet, l'augmentation du nombre de cœursinduit l'augmentation du nombre de messages.

Nous pouvons également, à partir de ces figures, constater le coût induit par la synchronisation des agents. En effet, la courbe uniquement basée sur des zones de recouvrement (OLZ) offre des temps d'exécution environ 8 fois meilleur pour 16 cœurs que la courbe basée sur la



(a) Speed-up

(b) Running time

FIGURE 4.3 – Extensibilité et temps d'exécution du modèle Virus (Jeu de valeurs 2 de la Table 4.2)

synchronisation décalée (SSD). Cette différence tend à décroitre avec l'augmentation du nombre de cœurs, par exemple, pour 256 cœurs cette différence n'est plus que de 6.6. En ce qui concerne la différence des temps d'exécution entre la courbe des écritures asynchrones (EA) et celle de la synchronisation décalée (SSD) elle n'est que d'un ratio de 1.27 pour 16 cœurs, et de 1.4 pour 512 cœurs.

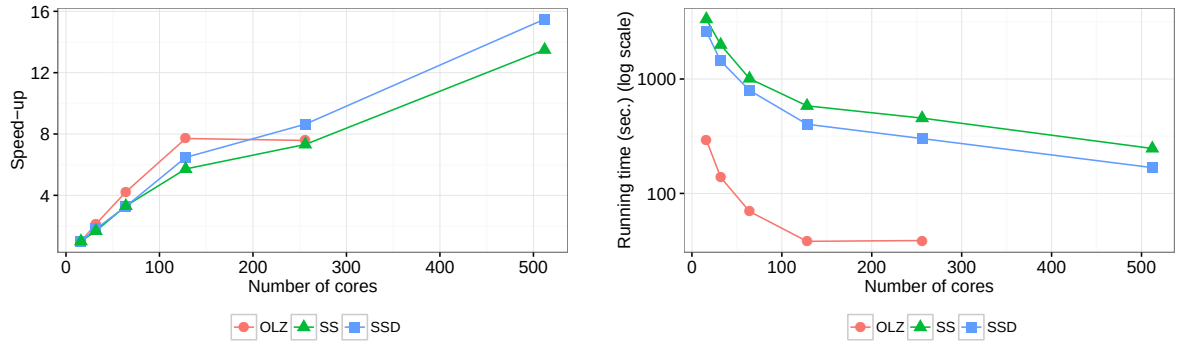
Modèle proie-prédateur

Le modèle proie-prédateur utilisé pour les expérimentations est basé sur une grille de 400x400 qui représente l'environnement où 25000 moutons et 17000 loups sont initialement positionnés de manière aléatoire. Comme pour les autres modèles, pour des soucis de reproductibilité, la même configuration initiale est utilisée pour toutes les exécutions des simulations, seule la graine varie d'une exécution à une autre. Le modèle utilisé pour les comportements et l'initialisation de l'énergie des agents loups et moutons est issu du modèle de la plateforme NetLogo [37]. L'énergie gagnée par un mouton lorsqu'il mange de l'herbe est fixée à 5 et à 20 lorsqu'un loup mange un mouton. En ce qui concerne les taux de reproduction, ils sont fixés à 5 pour les moutons et à 4 pour les loups. Pour finir les simulations sont exécutées durant 2000 pas de temps. La Table 4.3 présente les jeux de paramètres détaillés pour les expérimentations du modèle proie-prédateur.

Paramètres	Jeu de valeurs 1
Environnement	400x400
Nb. mouton	25000
Nb. loup	17000
Tx. reproduction mouton	0.5
Tx. reproduction loup	0.4
Gain nourriture mouton	4
Gain nourriture loup	20
Vie initiale mouton	4
Vie initiale loup	20
Croissance herbe	8
Nb. Pas de temps	2000

TABLE 4.3 – Jeux de paramètres pour les expérimentations du modèle proie-prédateur

La Figure 4.4 présente l'extensibilité 4.4(a) et les temps d'exécution 4.4(b) du modèle proie prédateur avec le jeu de valeurs 1 de la Table 4.3 pour les trois politiques de synchronisation *OLZ*, *SSD* et *SS*. Nous n'avons pas utilisé le modèle de synchronisation AS pour ce modèle car il ne correspond pas au besoin du modèle. En effet lorsqu'un loup mange un mouton il doit savoir si il l'a réellement mangé pour augmenter son énergie.



(a) Speed-up

(b) Running time

FIGURE 4.4 – Extensibilité et temps d'exécution du modèle proie-prédateur (Jeu de valeurs 1 de la Table 4.3)

L'analyse des courbes (Figure 4.4(a)) montrent que la plateforme est extensible pour le modèle proie-prédateur. Seule la courbe OLZ (ne gérant pas les écritures dans les zones de recouvrements) décroisse à partir de 128 cœurs. Cela s'explique par le fait que la taille du modèle est trop petite et que le temps d'exécution étant déjà très bas, la limite de l'extensibilité est donc atteinte. Pour cette raison, aucun calcul n'a été effectué sur 512 cœurs avec cette politique de synchronisation. En ce qui concerne les courbes SS et SSD, gérant les synchronisations, on remarque que les courbes s'inversent à partir de 64 cœurs par rapport au modèle Virus. La courbe SSD qui auparavant offrait de moins bons résultats que la courbe AS avec le modèle Virus, offre avec ce modèle de meilleurs résultats que la courbe SS. Nous expliquons cette inversion par le fait que, dans le modèle proie-prédateur, le comportement des agents loups et moutons est dépendant de la synchronisation. C'est à dire, que suivant le résultat de la synchronisation l'agent va modifier son état interne en augmentant sa durée de vie ou non. De ce fait, la synchronisation étant bloquante dans la politique SS, plus de temps est consommé et donc de moins bonnes performances sont observées.

Synthèse

D'une manière générale, sur les trois modèles étudiés, les courbes de speed-up montrent toutes une progression au moins linéaire, ce qui permet de conclure à une bonne extensibilité. Les speed-ups sont calculés relativement à des mesures ayant déjà un certain degré de parallélisme, 16 cœurs. De ce fait ces courbes sont relativement similaires quelque soit le niveau de synchronisation utilisé. Le niveau de synchronisation choisi pour implémenter le modèle n'a donc pas d'impact sur l'extensibilité.

Il peut paraître surprenant que certaines courbes obtiennent un speed-up supra-linéaire (c'est le cas pour le modèle Flocking et, dans une moindre mesure pour le modèle virus) mais cela signifie que les simulations avec un grand nombre de cœurs bénéficient mieux du parallélisme. Ceci est plutôt encourageant en terme de parallélisme et confirme que les systèmes multi-agents sont de bons candidats à une parallélisation.

Les courbes étudiées dans cette partie mettent en évidence que les politiques de synchronisation engendrent un coût important à cause du volume de communications nécessaire à la synchronisation. Néanmoins ce coût n'augmente pas avec le nombre de cœurs utilisés pour la simulation. Ceci peut être expliqué par le fait que les interactions n'ont lieu qu'entre voisins dans les modèles étudiés. Or, quelque soit le nombre de cœurs utilisés, le nombre de voisins reste constant pour chacun des processus. Le nombre de zones de recouvrement à gérer est donc constant. Cependant, le nombre d'agents présents dans une zone de recouvrement peut avoir un impact sur le nombre de synchronisations à réaliser. Nous étudions donc la montée en charge des différentes politiques de synchronisation dans la section suivante.

4.4.2 Analyse de la montée en charge

Pour calculer la montée en charge nous fixons le nombre de processus et nous faisons varier le nombre d'agents. Dans cette section, uniquement les modèles Flocking et Virus sont présentés. Le modèle proie prédateur n'est pas présenté car il est très difficile de faire varier le nombre d'agents étant donné que dans ce modèle la population qui compose la simulation s'auto-équilibre. Pour les deux modèles (Flocking et Virus), les même politiques de synchronisation sont expérimentées, afin d'observer leurs comportements face à la montée en charge.

Modèle Flocking

Le jeu de valeurs utilisé pour évaluer la montée en charge du modèle Flocking est le même que celui présenté dans Table 4.1, à la différence que le nombre d'agents oiseaux qui composent la simulation varie de 100000 à 1000000. La Figure 4.5 présente la montée en charge du modèle Flocking utilisant 512 cœurs. Il faut noter que dans la littérature, peu d'articles reportent des exécutions avec au moins 1 million d'agents. Quelques expériences dont celle de RepastHPC [93], simulent environ 68 billions d'agents mais avec plus de 32700 processus. Le modèle utilisé pour atteindre ce nombre d'agents est le modèle "Triangle". Ce modèle consiste à déplacer des agents en fonction d'un agent de référence, les comportements des agents sont donc très simples. Dans notre cas, nous arrivons à simuler des modèles avec des comportements plus élaborés.

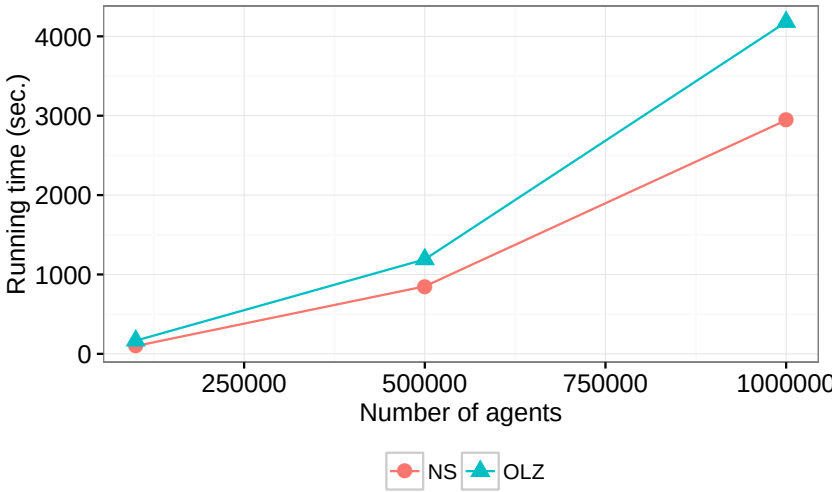


FIGURE 4.5 – Montée en charge du modèle Flocking de 100000 à 1000000 agents utilisant 512 cœurs et 2000 pas de temps

La figure 4.5 montre qu'une simulation avec un million d'agents peut être exécutée en environ une heure sur notre système. Ce qui laisse supposer qu'il est possible de faire tourner des simulations de bien plus grande taille sur cette plateforme.

On constate que les deux implémentations (NS et OLZ) supportent bien la charge jusqu'à 500000 agents, au delà de 500000 agents, les courbes croissent plus rapidement. Sans surprise, la version sans synchronisation, ni zone de recouvrement supporte mieux la charge que la version synchronisée qui consomme environ un tiers de performance en plus. Nous expliquons cet accroissement de la charge à partir de 500000 agents par un manque d'optimisation lors de la recherche du voisinage des agents. En effet, un parcours du champ de perception est effectué à chaque pas de temps au lieu de mettre à jour uniquement les agents qui arrivent ou partent du voisinage.

Modèle Virus

Les paramètres utilisés pour analyser la montée en charge du modèle Virus sont les mêmes que ceux présentés dans Table 4.2 jeu de valeurs 2. Dans ce modèle, pour faire varier la charge, nous faisons varier la capacité totale que la simulation accepte de 100000 à 700000 agents mobiles.

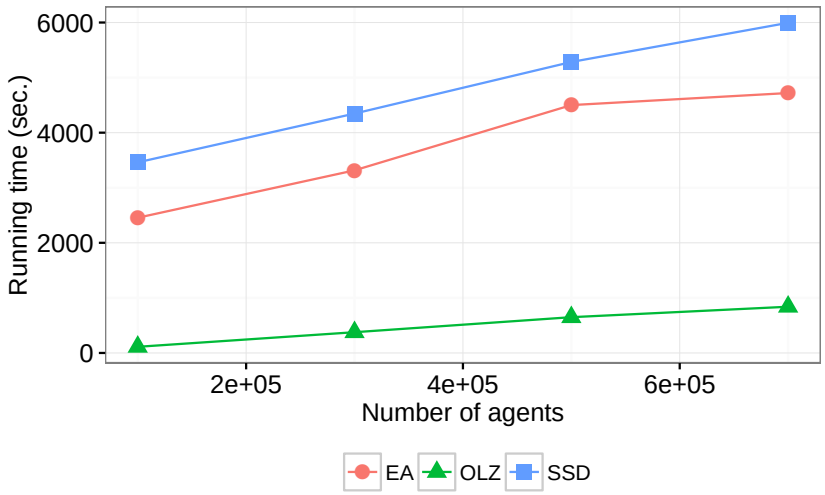


FIGURE 4.6 – Montée en charge du modèle Virus de 100000 à 700000 agents utilisant 64 cœurs et 800 pas de temps

La figure 4.6 montre la montée en charge du modèle Virus de 100000 à 700000 agents utilisant 64 cœurs et 800 pas de temps. Il peut paraitre surprenant de n'utiliser que 64 cœurs pour effectuer les expérimentations de montée en charge du modèle Virus mais ces expérimentations ont été réalisées avant celles du modèle Flocking et obtenir autant de ressources de calcul (512 cœurs) n'est pas simple sur un cluster partagé avec plusieurs utilisateurs. Pour cette raison seulement 64 cœurs sont utilisés pour les expérimentations.

Parmi les courbes, celle qui supporte mieux la charge est évidemment la courbe ne gérant pas les écritures dans la zone de recouvrement (OLZ). En ce qui concerne les courbes qui gèrent les écritures dans les zones de recouvrement, on constate que l'écriture asynchrone (AS) supporte mieux la charge que la synchronisation forte décalée (SSD). En effet, la courbe SSD, reste linéaire, alors que la courbe AS croît très peu de 500000 à 700000 agents. La charge est moins bien supportée par la SSD à cause de l'accumulation des synchronisations en fin de pas de temps qui ajoutent un surcout à la simulation.

Sur cette figure nous pouvons constater que le ratio des résultats obtenus entre 100k et 700k pour la courbe AS est de 1,9, 1.73 pour SSD et 7.5 pour OLZ.

Les résultats sur l'extensibilité et la montée en charge montrent que, quelque soit le niveau de synchronisation utilisé, les modèles multi-agents sont de bons candidats à la parallélisation. Les résultats obtenus par le niveau de synchronisation OLZ, uniquement basé sur un échange des zones de recouvrement, montrent que ce niveau que les autres niveaux plus contraints. De ce fait nous nous sommes interrogés sur l'impact du niveau de synchronisation, l'erreur engendrée, sur les résultats du modèle. Nous présentons cet impact dans la suite.

4.4.3 Analyse de l'impact de la synchronisation sur les résultats

Pour étudier l'impact des synchronisations sur les résultats des simulations nous avons observé deux résultats : le résultat de la simulation lui-même et le nombre d'interactions fausses. Le résultat d'une simulation est ce qui est attendu par le modélisateur. Il est donc naturel de vérifier l'impact d'une exécution plus ou moins synchronisée sur le résultat attendu par le modélisateur. Par ailleurs, dans le cas où les écritures dans les zones de recouvrement sont gérées, c'est à dire avec les politiques AS, SS et SSD, des interactions peuvent échouer, c'est à dire rendre un résultat faux. Pour analyser l'impact des interactions sur les résultats des simulations nous comptabilisons le nombre total de demandes de synchronisation durant les exécutions des simulations puis nous détaillons les synchronisations qui ont abouti et celles qui ont échoué

Pour cette analyse nous n'avons pas inclu le modèle Flocking. En effet, les résultats de ce modèle sont visuels et ne peuvent donc pas être inclus dans un rapport pour étudier l'impact d'une politique. Dans la suite, nous présentons, dans un premier temps, les résultats pour le modèle Virus et dans un deuxième temps, ceux pour le modèle proie-prédateur. Dans les expériences réalisées nous notons les interactions qui échouent comme *FALSE* et à l'inverse les interactions qui n'échouent pas comme *TRUE*.

Modèle Virus

La Figure 4.7 présente l'ensemble des demandes de synchronisation pour le modèle virus avec le jeu de valeurs 2 de la Table 4.2.

On constate qu'entre les deux politiques de synchronisation le nombre total de demandes de synchronisation croît de manière linéaire avec le nombre de cœurs. Cela s'explique par le fait que la simulation est divisée en plus de cellules, qu'il y a donc plus de zones de recouvrement, et plus d'interactions potentielles. On constate aussi que le nombre d'appels *FALSE* est très faible pour la politique AS. A l'inverse pour la politique SSD le nombre d'appels *FALSE* est plus important d'environ 10%. Nous expliquons cette différence par le fait que les synchronisations sont gérées en fin de pas de temps. En effet, les agents qui effectuent une demande de synchronisation sont suspendus et sont exécutés lors de la réception de la réponse en fin de pas de temps. De ce fait, certaines interactions qui étaient potentiellement *TRUE* au moment de la demande de synchronisation peuvent devenir *FALSE* si les agents concernés n'avait pas encore été exécutés. Nous pouvons nous poser la question de savoir quel est l'impact de ces interactions *FALSE* sur les résultats.

La Figure 4.8 présente les résultats d'une même exécution sur 128 cœurs avec les différentes politiques de synchronisation. Nous constatons que les Figures 4.8(a) (politique SSD) et 4.8(b) (politique AS) offrent des résultats quasi identiques malgré quelques variations qui s'expliquent par les interactions *FALSE* traitées en fin de pas temps par la politique SSD. En revanche, la Figure 4.8(c) (politique OLZ qui est sans gestion des écritures dans les zones de recouvrement)

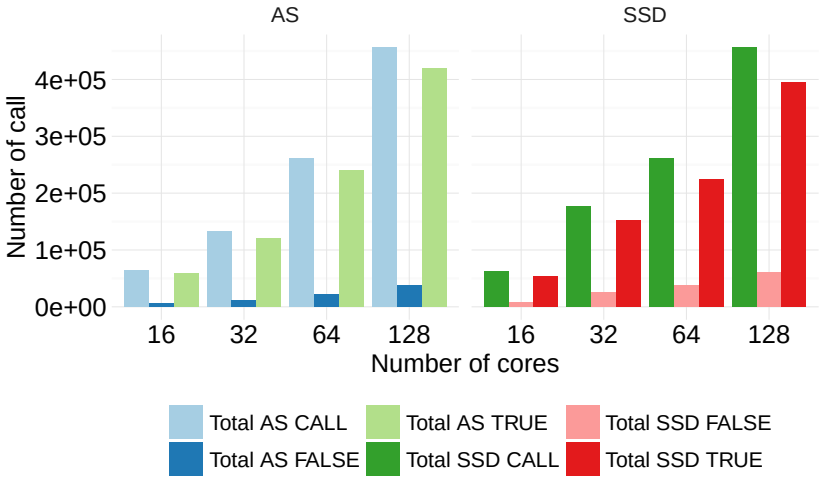


FIGURE 4.7 – Détail des demandes de synchronisations du modèle Virus avec le jeu de valeurs 2 de la Table 4.2 pour les politiques de synchronisation AS et SSD

offre une tendance identique, mais de nombreuses fluctuations par rapport aux deux autres courbes sont constatées.

L'influence des politiques de synchronisation sur les résultats est encore plus marquée avec le modèle proie-prédateur que nous présentons dans la partie suivante.

Modèle proie-prédateur

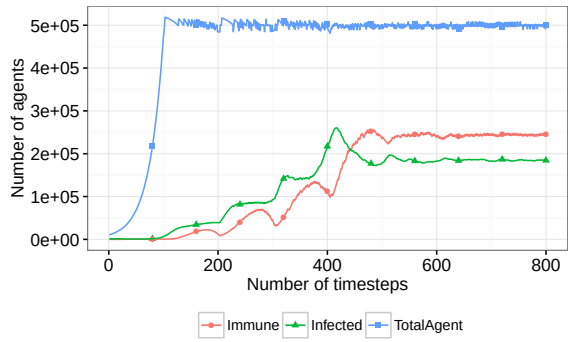
Comme pour le modèle Virus, la Figure 4.9 présente l'ensemble des demandes de synchronisation pour le modèle proie-prédateur avec le jeu de valeurs 1 de la Table 4.3.

De la même manière que pour le modèle Virus, on constate qu'entre les deux politiques de synchronisation le nombre total de demandes de synchronisation croît de manière linéaire plus il y a de cœurs. On constate aussi que le nombre d'appels *FALSE* reste plus faible pour la politique SS que pour la politique SSD.

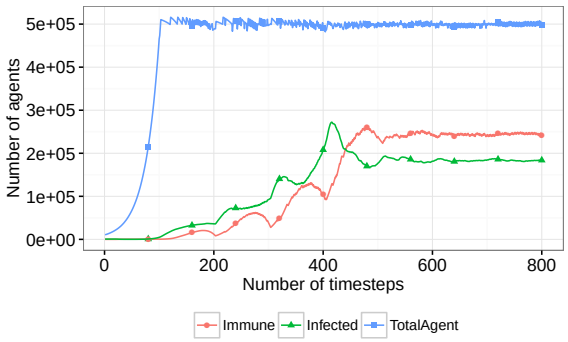
La Figure 4.10 présente les résultats d'une exécution sur 128 cœurs avec les différentes politiques de synchronisation. Nous constatons que la Figure 4.10(a) n'est pas semblable à la courbe de la Figure 4.10(b). Ceci est du à l'instabilité du modèle. Comme nous l'avons dit précédemment, le modèle proie-prédateur est un modèle très sensible : la modification d'une des données d'entrée peut conduire à une instabilité du modèle qui se traduit par la mort d'une des espèces.

Les résultats expliquent, la raison pour laquelle la politique de synchronisation SSD offre de meilleurs résultats de performance que la politique SS pour ce modèle. Comme de nombreuses interactions ne s'effectuent pas car elles sont traitées en fin de pas de temps, la simulation est de fait modifiée et offre des résultats erronés. Il est important de noter que l'on retrouve quand même des courbes cycliques qui tendent à s'équilibrer.

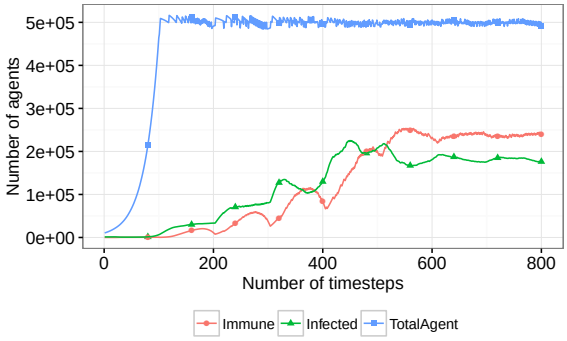
Contre toute attente, la Figure 4.10(c) qui représente l'exécution sans synchronisation offre au premier abord des résultats plus proches que la politique de synchronisation SSD (Figure 4.10(a)). Au vue des courbes, on peut s'interroger sur les résultats du modèle ainsi que sur leurs portées, à savoir, qu'est ce qui est important d'observer. En effet, si nous regardons plus en détails les Figures 4.10(c) et 4.10(a) concernant respectivement les résultats des courbes sans synchronisation (OLZ) et avec synchronisation forte décalé (SSD), nous remarquons que la valeur maximale obtenue pour les moutons est au alentour de 60000, tandis que sur la Fi-



(a) SSD



(b) AS



(c) OLZ

FIGURE 4.8 – Résultats d'une exécution du modèle Virus utilisant 128 cœurs avec trois politiques de synchronisation (Jeu de valeurs 2 de la Table 4.2)

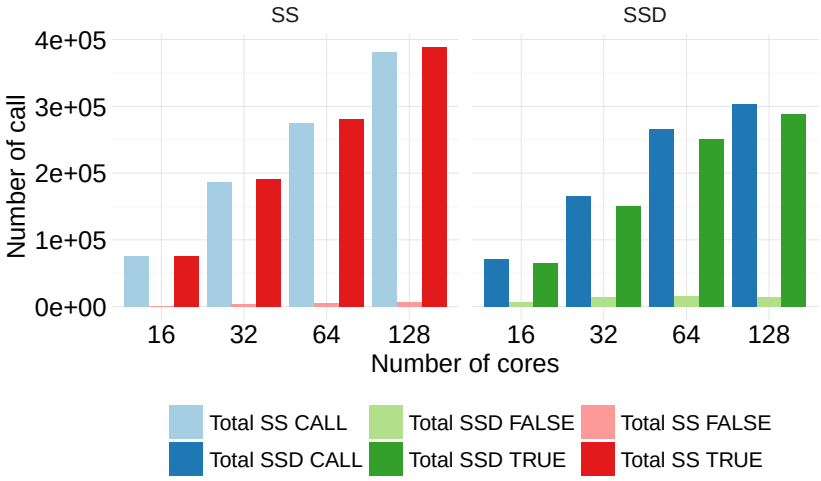
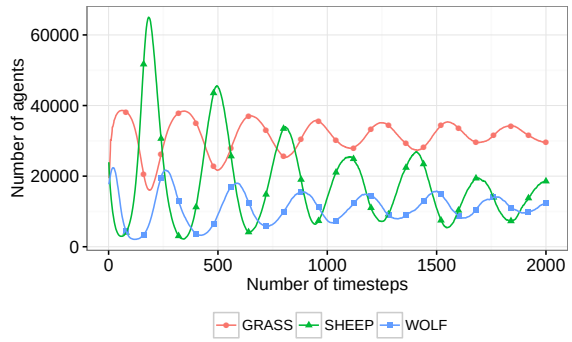
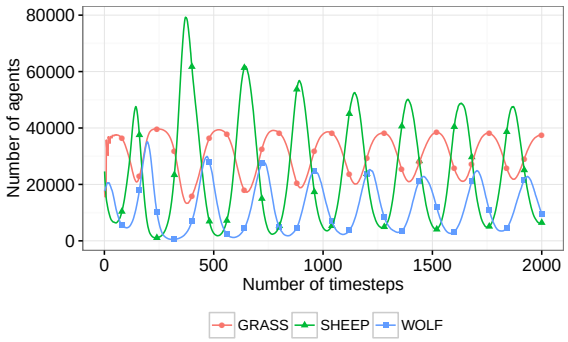


FIGURE 4.9 – Détail des demandes de synchronisations pour le modèle proie-prédateur avec le jeu de valeurs 1 de la Table 4.3

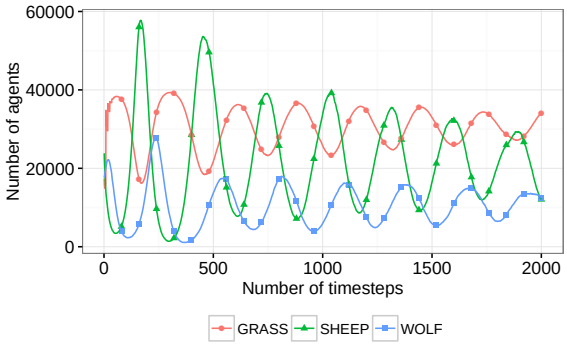
gure 4.10(b) synchronisation SS, la valeur maximale obtenue par les moutons est d'environ 80000.



(a) SSD



(b) SS



(c) OLZ

FIGURE 4.10 – Résultats d'une même exécution du modèle Virus utilisant 128 cœurs avec trois politiques de synchronisation (Jeu de valeurs 2 de la Table 4.2)

Pour finir, on remarque que sur la Figure 4.10(b), c'est à dire avec synchronisation forte (SS) la phase est régulière avec un décalage de l'ordre de $\pi/2$. En revanche, sur la Figure 4.10(c), un décalage de phase s'effectue au cours du temps. Par conséquent, les courbes sans synchronisation (OLZ) et avec synchronisation décalé (SSD) sont semblables et erronées.

4.5 Discussion

Après avoir présenté les résultats d'expérimentations et analysé l'impact de la synchronisation sur les résultats de simulation et d'exécution nous donnons dans cette partie quelques réflexions issues de notre expérience.

Notre étude sur l'impact des politiques de synchronisation vise à sensibiliser la communauté agent, et plus particulièrement les modélisateurs, aux problèmes liés à la parallélisation d'un modèle. En effet, à travers nos expérimentations, nous quantifions le coût de la synchronisation dans la simulation suivant les modèles qui sont animés. Ce coût de synchronisation, non négligeable, est important à prendre en compte lors de la modélisation d'un modèle en parallèle. Suivant le modèle que l'on souhaite implémenter, le niveau de synchronisation peut être adapté afin d'obtenir les meilleures performances possibles lors de l'exécution. C'est la raison pour laquelle, nous avons défini différentes politiques de synchronisation qui permettent de gérer différents types de modèles.

Comme la synchronisation est de fait dépendante des comportements des agents, c'est à

la charge du modélisateur de prêter attention à la manière dont s'exécutera le modèle et donc d'adapter la modélisation. Par exemple, dans le cas du modèle proie-prédateur, le modélisateur a une responsabilité importante car, lors de l'étape de modélisation, il a 2 choix : soit les prédateurs peuvent manger les proies situées sur les cellules voisines, soit les prédateurs mangent uniquement les proies situées sur leur cellule. Nous avons implémenté le premier choix qui implique l'utilisation de la synchronisation forte. Avec le second choix, puisque les loups ne mangent que dans la cellule locale, il n'y a pas de concurrence d'accès en écriture dans les zones de recouvrement, les performances seront meilleures. Mais on peut se poser la question de savoir, si cette modélisation reste valide et quel est son impact sur les résultats de la simulation. Comme nous proposons des mécanismes pour les plateforme agents parallèle, la démarche de modélisation reste donc à la charge du modélisateur, et lui permet de savoir s'il est prêt à payer le surcoût de synchronisation au dépend d'une autre manière de modéliser. D'une manière plus générale, la réflexion sur l'implémentation des modèles pose la question qui est de savoir si tous les modèles peuvent être conçus sans écriture.

Chapitre 5

Gérer la communication entre agents

Sommaire

Dans les chapitres précédents nous n'avons utilisé la communication au sein de notre plateforme que comme un moyen pour distribuer la charge ou pour synchroniser des agents. Cependant un certain nombre de modèles nécessitent de proposer des fonctions de communication au niveau des agents. La communication est alors une fonctionnalité de l'agent, accessible au modélisateur pour implémenter son modèle. Parmi les modèles que nous avons présentés, le modèle de référence utilise par exemple la communication directe entre agents, sans passer par un support intermédiaire comme l'environnement. D'une manière plus générale, plusieurs types de modèles peuvent utiliser des fonctions de communication, parmi ceux-ci nous pouvons citer :

— les modèles orientés mobilité. Par exemple, en mobilité urbaine, les agents peuvent avoir besoin de continuer à communiquer avec leurs contacts tout en se déplaçant. En raison de la répartition géographique ou non de la simulation, les agents se déplacent n'importe où sur l'environnement et donc sur les différents processus. Cependant, ils doivent toujours être en mesure de pouvoir communiquer avec les agents qui étaient à proximité durant une partie de la simulation.

— les modèles orientés graphes ou réseau. Limiter la communication d'un agent à son propre voisinage conduit à des contraintes de distribution complexes : par exemple, si le graphe n'est pas planaire il est très difficile de le distribuer correctement et efficacement entre les différents processus.

Malheureusement, la fonctionnalité de communication distante, entre tous les processus qui exécutent une simulation multi-agents, n'est proposée que par la plateforme Flame. Les autres plateformes se limitent aux communications sur le processus local. La mise en place de cette fonctionnalité n'est en effet pas si triviale qu'il n'y paraît et nécessite une réflexion pour être implémentée dans une simulation s'exécutant sur des super-calculateurs.

Dans ce chapitre, nous étudions les problématiques posées par l'introduction de la fonctionnalité de communication dans les simulation multi-agents s'exécutant sur des ressources HPC. Nous proposons un schéma de communication spécifique que nous validons en l'implémentant dans la plateforme FractalPMAS ainsi qu'une étude de performance sur un super-calculateurs.

5.1 Problématique

Dans un contexte d'exécution HPC, c'est-à-dire sur un super-calculateur, la mise en place de la communication entre agents distants induit certaines difficultés. Nous abordons ici les contraintes induites ce contexte d'exécution spécifique, qui impose de n'avoir qu'un seul processus par cœur alloué, et par la distribution des agents sur des processus différents, qui impose une gestion stricte de la délivrance des messages pour tendre vers des simulations reproductibles.

5.1.1 Communication entre agents dans un contexte HPC

Le fait de cibler le calcul haute performance implique certaines contraintes sur l'implémentation des MAS. Pour rappel, habituellement, les plateformes PDMAS sont implémentées en respectant le paradigme Single Process Multiple Data (SPMD), afin de permettre une meilleure extensibilité. Les simulations multi-agents impliquent généralement plusieurs dizaines de milliers d'agents qui potentiellement interagissent entre eux par le biais de la communication à chaque pas de temps de simulation. La communication est donc le point clef mais aussi le goulet d'étranglement dans un contexte parallèle. Évidemment, le temps d'exécution est impacté par la fréquence des communications lors de l'exécution de la simulation.

Dans le contexte parallèle, le standard de communication pour les infrastructures de type Cluster HPC est Message Passing Interface (MPI). Il est important de faire attention aux primitives de communications que nous utilisons afin de réduire au maximum le coût des communications. Malheureusement, le gestionnaire Batch SGE ne permet pas de gérer le fait d'allouer plusieurs processus à un même cœur. En effet, avec SGE les processus sont alloués sur plusieurs *nœuds* sans avoir le contrôle du placement sur chaque *nœud* de fait, cela rend difficile l'utilisation de plusieurs processus. Pour cette raison, un seul processus est alloué par cœur. En revanche, les gestionnaires SLURM et PBS permettent d'allouer plusieurs processus par cœur (PPN = Process Per Node), de fait, il est donc possible d'exécuter des applications Hybrides (MPI + OpenMP).

Le fait de n'utiliser qu'un seul un processus par cœur, ne nous permet pas d'avoir recours à des mécanismes de communication comme les *listener*, *onEvent* ou encore *onMessage* pour communiquer qui dédient un thread uniquement pour attendre les messages pendant que les agents sont exécutés. Les messages peuvent donc être reçus de deux manières différentes, soit par réception non bloquante durant l'exécution des agents soit par réception bloquante à la fin de l'exécution de chaque pas de temps. Cela illustre la complexité d'utiliser un seul processus pour exécuter un modèle qui implémente des communications asynchrones.

5.1.2 Délivrance des messages

Un autre problème posé par le contexte parallèle est le problème de livraison des messages dans une simulation multi-agents par pas de temps. Pour rappel, dans les simulations par pas de temps, contexte dans lequel nous nous situons, la simulation est divisée en intervalles de temps qui représentent la discrétisation temporelle de la simulation. Les messages entre agents sont ainsi délimités par ces pas de temps. La question est : à quelle étape de temps le message doit être délivré ?

Comme abordé précédemment dans le Chapitre 3 à la Section 3.7.1, pour garantir la reproductibilité de la simulation, chaque message envoyé au pas de temps n doit être reçu avant le début du pas de temps $n+1$. Comme les processus s'exécutent de manière asynchrone les uns des autres et que l'ordre d'exécution des agents dans un processus est aléatoire, la délivrance des messages dans le même pas de temps n ne peut être garantie. En effet, l'agent destinataire du message peut avoir été exécuté, pour le pas de temps n, avant l'émission du message. La délivrance des messages au début du pas de temps $n+1$ est alors le seul moyen de garantir qu'un message sera toujours reçu au même pas de temps quelque soit l'ordre d'exécution des agents. Si les pas de temps ne sont pas respectés alors de potentiels erreurs peuvent être injectées dans la simulation et la reproductibilité ainsi que la cohérence sont impossibles à garantir.

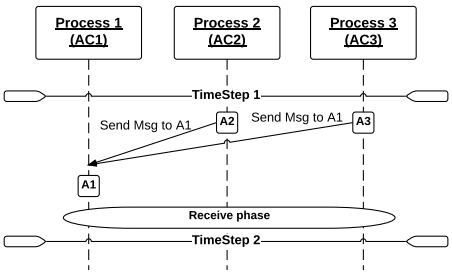


FIGURE 5.1 – Indéterminisme dans l'ordre des messages

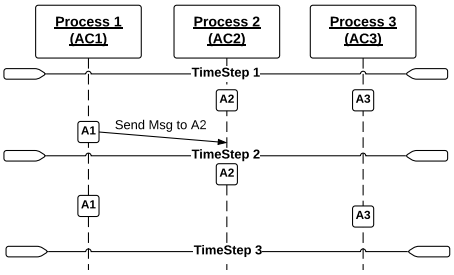


FIGURE 5.2 – Indéterminisme dans le pas de temps

Il est important de noter que même avec la délivrance des messages au pas de temps $n+1$, la nature stochastique des simulations multi-agents fait qu'il est difficile de fournir une garantie absolue lors de la simulation. Nous avons illustré ce problème sur les figures 5.1 et 5.2.

La figure 5.1 montre un cas d'indéterminisme de réception des messages. Sur cette figure nous prenons, à titre d'exemple, P_1, P_2 et P_3 trois processus exécutant une partie d'une simulation. Si les deux processus P_2 et P_3 envoient un message en même temps à destination du processus P_1, alors nous ne connaissons pas l'ordre dans lequel nous devons les délivrer sur P_1. Ce problème est énnoncé par Lamport [94] pour les systèmes distribués. Cet indéterminisme ne se pose pas dans le cas d'un système centralisé car l'exécution est séquentielle. Les messages sont donc traité un à un.

La figure 5.2 illustre la nécessité d'avoir des phases de réception entre les pas de temps. Par exemple si l'agent $A1$ est exécuté à la fin du pas de temps et envoie un message à l'agent $A2$. Si le message est retardé sur le réseau et que l'agent $A2$ est exécuté au tout début du pas de temps suivant alors l'agent $A2$ peut ne pas recevoir le message qui lui était destiné au cours de ce pas de temps. Pour cette raison, il est important de définir des phases de réception à la fin de chaque pas de temps.

5.2 Schéma de communication

A partir de ces réflexions, nous proposons un schéma de communication pour PDMAS afin de permettre les communications locales mais aussi distantes entre agents sans prêter attention à l'emplacement d'exécution de l'agent. Notre schéma de communication permet la reproductibilité et garantit que chaque message envoyé au pas de temps n est reçu au début du pas de temps $n+1$. Cet algorithme se décompose en deux phases, dans un premier temps une phase de réception des messages et dans un deuxième temps, après avoir présenter le système d'identification utilisé pour les agents, nous introduisons la notion de proxy qui nous permet de garder une trace de l'emplacement où s'exécute un agent.

5.2.1 Phase de réception des messages

Pour des raisons de cohérence et de reproductibilité, chaque message envoyé au pas de temps n doit être délivré à l'agent destinataire au début du pas de temps suivant $n + 1$. Pour qu'un processus passe au pas de temps suivant il doit donc avoir reçu tous les messages qui ont été émis dans le pas de temps courant à destination de ses agents. En raison de la nature stochastique des agents, il n'est pas possible de savoir, à la fin d'un pas de temps, combien de messages doivent être reçus par un processus et à partir de combien de processus. Il est alors impossible de savoir quand un processus est prêt pour exécuter le pas de temps suivant. Il faut donc synchroniser les processus afin d'être sûr que chacun des processus est prêt pour exécuter le pas de temps suivant.

Pour synchroniser des processus, MPI propose des mécanismes comme des *barrier* qui sont des points de synchronisation, mais ils ne résolvent pas ce problème. En effet, si nous utilisons des *barrier* à la fin de chaque pas de temps pour synchroniser les processus, alors certains processus atteignent le point de synchronisation plus rapidement que les autres et restent bloqués, inactifs, jusqu'à ce que les autres processus le rejoignent. Cependant, les processus qui n'ont pas encore atteint le point de synchronisation peuvent encore envoyer des messages à destination des agents gérés par des processus déjà bloqués dans la *barrier*. Dans ce cas, les processus bloqués dans le point de synchronisation ne pourront pas recevoir ces messages et ceux-ci seront perdus. Pour ces raisons, les mécanismes proposés par MPI ne sont malheureusement pas suffisants pour résoudre le problème.

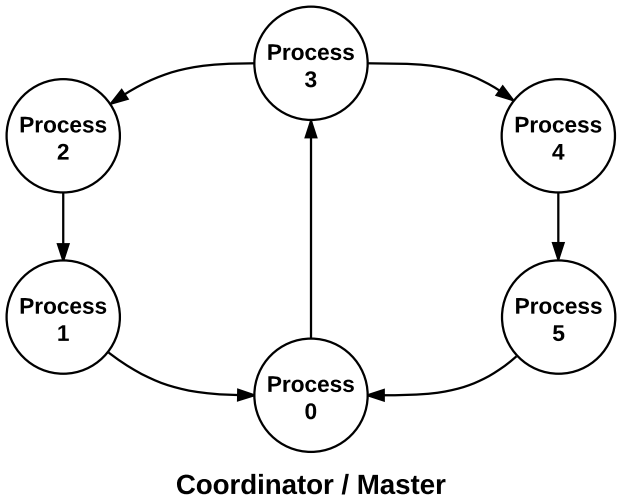


FIGURE 5.3 – Schéma d'un anneau bi-directionnel de terminaison

Afin de surmonter ce problème, nous utilisons un algorithme de terminaison qui va permettre de trouver un accord entre les différents processus afin d'exécuter le pas de temps suivant. Quand tous les processus ont terminé les différentes étapes alors le pas de temps suivant peut être exécuté. L'algorithme de terminaison que nous utilisons est basé sur un anneau bi-directionnel avec coordinateur comme présenté dans la Figure 5.3. Le fonctionnement est le suivant, chaque processus envoie un message au coordinateur de l'anneau lorsqu'il a terminé son traitement. Lorsque le coordinateur a reçu tous les messages de terminaison de tous les processus (excepté lui-même), alors il envoie un message de fin à un processus. Ce dernier est choisi tel que le rang soit égal à la moitié du nombre total de processus. Ensuite le processus destinataire du message du coordinateur envoie deux messages, l'un au processus de rang supérieur et l'autre au processus de rang inférieur. Ce double envoie de messages permet d'initier les deux branches de l'anneau.

Pour finir, chaque processus destinataire envoie un message au processus de rang supérieur ou inférieur, suivant la branche de l'anneau dans laquelle il se situe jusqu'à ce que le coordinateur reçoive les messages. Cet algorithme de terminaison nous permet d'être sûr que tous les processus ont effectivement terminé de recevoir les messages des autres processus. L'anneau bi-directionnel est choisi pour des raisons d'efficacité comparé à un simple anneau unidirectionnel.

Dans notre proposition de schéma de communication, les agents envoient directement les messages durant l'exécution de leur comportement de manière non-bloquante en utilisant des fonctions de communication que nous proposons comme fonctionnalités de base d'un agent. La fonction d'envoi d'un message est de la forme *communicate(MSG)* avec comme paramètre le message qui est un objet sérialisable composé du destinataire, de l'expéditeur et du contenu du message. Les agents ne reçoivent pas directement les messages de réponse. En effet, ce sont les AC qui sont responsables de la réception des messages pour le compte de leur agents concernés. Le fonctionnement est simple, pour être sûr que chaque message envoyé au cours du pas de temps est bien reçu, les AC acquittent (*Ack*) chaque message reçu de chaque agent pour être sûr qu'il n'y a pas de message en attente avant de passer au pas de temps suivant. L'acquittement de tous les messages est nécessaire pour garantir qu'il n'y a pas de messages en transit. Cela permet d'éviter le cas où tous les processus sauf un sont en attente pour exécuter le pas de temps suivant. Si le processus qui n'est pas en attente pour passer au pas de temps suivant envoie un message puis signale qu'il a finit alors tous les processus passe au pas de temps suivant mais un message sera toujours en transit et il ne sera pas traité au bon pas de temps et donc ce message peut potentiellement injecter des incohérences dans la simulation. La Figure 5.4 représente un schéma de diffusion de messages exécuté entre chaque pas de temps.

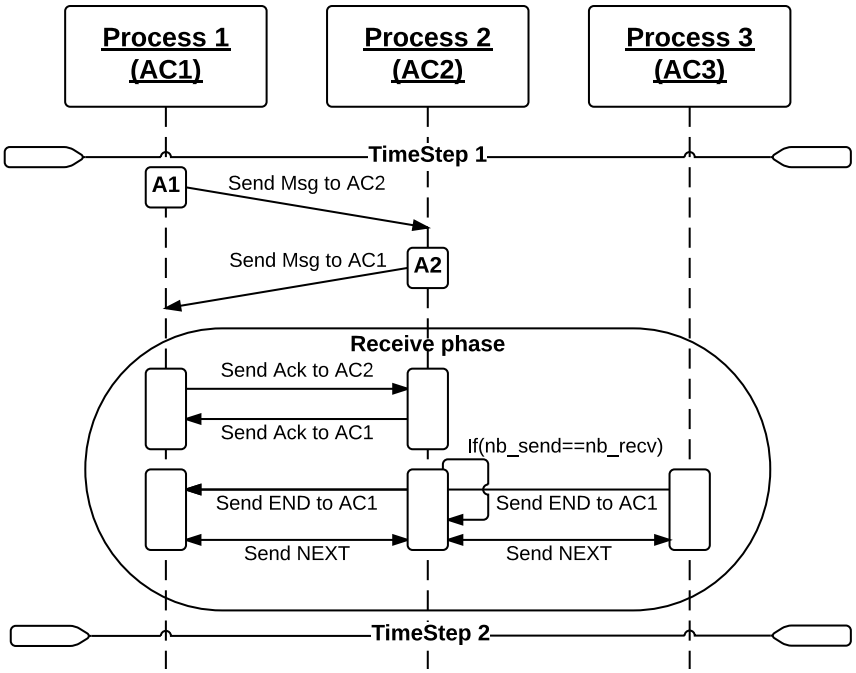


FIGURE 5.4 – Schéma de diffusion de messages exécutés entre chaque pas de temps

L'algorithme 2 détaille la phase de réception des messages exécutée par chaque AC à chaque pas de temps. L'algorithme se décompose en deux parties, une partie exécutée par le coordinateur de la ligne 1 à la ligne 12 et l'autre exécutée par les autres processus (ligne 13 à la ligne 29). Dans les deux parties, les processus attendent l'acquittement de tous les messages envoyés par les agents dont les AC sont responsables (lignes 2 et 13). Ensuite, les AC envoient un message

de fin END au un coordinateur (ligne 21). Le coordinateur est généralement le processus de rang 0. Lorsque le coordinateur a reçu tous les messages de fin END des autres processus, il débute alors l'algorithme de terminaison (ligne 10), pour cela il envoie un message $NEXT$ en utilisant l'anneau bi-directionnel puis attend le retour du message $NEXT$ (ligne 11). Les autres processus exécutent les lignes 22 à 28 pour transmettre le message de fin END. A la fin de l'algorithme tous les processus peuvent exécuter le pas de temps suivant.

Algorithme 2 : Phase de réception des messages exécuté par chaque AC à chaque pas de temps

```
1  if (AC is coordinator) then
2  |  while ((NbSendMessage ≠ NbAckReceived) && (nbEndReceived ≠ NbProcesses-1)) do
3  |  |  RecvMsg(Msg)
4  |  |  switch Msg.TAG do
5  |  |  |  case DATA_MSG do DeliverApplyMSG(Msg) ; SendAck(Msg.Source);
6  |  |  |  case ACK do NbAckReceived++;
7  |  |  |  case END do nbEndReceived++;
8  |  |  end switch
9  |  end while
10 |  SendNextTimestep(⌈NbProcesses/2⌉);
11 |  WaitForTermination();
12 else
13 |  while ((NbSendMessage ≠ NbAckReceived) && (NextTimestep ≠ false)) do
14 |  |  RecvMsg(Msg);
15 |  |  switch Msg.TAG do
16 |  |  |  case DATA_MSG do DeliverApplyMSG(Msg) ; SendAck(Msg.Source);
17 |  |  |  case ACK do NbAckReceived++;
18 |  |  |  case NEXT do NexTimestep ← true;
19 |  |  end switch
20 |  end while
21 |  SendEndToCoordinator(END);
22 |  if (AC.ID = ⌈NbProcesses/2⌉) then
23 |  |  SendNextTimestep(PREVIOUS_AC);
24 |  |  SendNextTimestep(NEXT_AC);
25 |  else if (AC.ID > ⌈NbProcesses/2⌉) then
26 |  |  SendNextTimestep(NEXT_AC);
27 |  else
28 |  |  SendNextTimestep(PREVIOUS_AC);
29 |  end if
30 end if
```

L'algorithme 2 fonctionne très bien pour les agents non mobiles. En revanche, dans les systèmes multi-agents, les agents sont souvent mobiles, ils ne sont pas fixés sur l'environnement, mais s'y déplacent. Ceci est, par exemple, le cas dans les modèles que nous avons définis au Chapitre 1 à la section 1.1.5. Dans le cas d'agents mobiles, les agents peuvent donc changer de processus AC. Par ailleurs, la distribution et la répartition dynamique de la charge sur les cœurs d'exécution impliquent que des agents sont migré d'un processus à l'autre. Lors de l'envoi d'un message à un agent, l'AC de l'émetteur a besoin de connaitre le numéro du processus sur lequel l'agent cible est exécuté afin de lui délivrer le message. Pour cela la mise en place d'un système d'identification des agents et d'un proxy qui permet de transférer les messages à un agent sont nécessaires. Nous présentons l'identification des agents et le fonctionnement des proxys, ainsi que l'algorithme utilisé pour résoudre ce problème, dans les sections suivantes.

5.2.2 Identification des agents

Dans les simulations multi-agents, un agent se doit d'être identifiable. Dans les PDMAS les agents doivent être identifiables indépendamment du processus sur lequel ils s'exécutent puisqu'ils sont à même de changer de processus au cours de la simulation.

Afin de pouvoir identifier à tout moment un agent, nous introduisons un système d'identification, largement inspiré de la plateforme RepastHPC et présenté dans la Figure 5.5. Ce système d'identification associe quatre valeurs à chaque agent afin de former un identifiant unique :

— Un identifiant global (GID),

— L'identifiant du processus qui a créé l'agent,

— L'identifiant du processus sur lequel l'agent s'exécute,

— L'identifiant du type de l'agent.

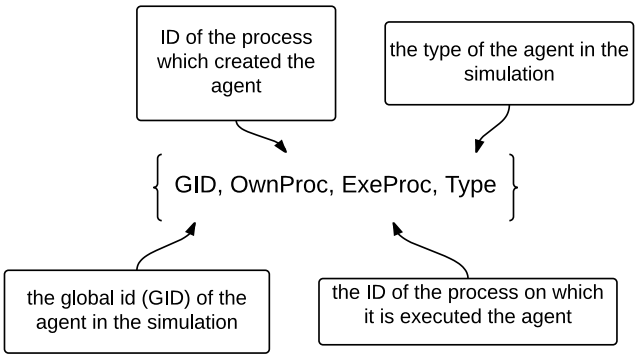


FIGURE 5.5 – Schéma de l'identifiant d'un agent

Avec ce système d'identification il est possible de connaitre à n'importe quel moment où un agent a été créée et où il s'exécute. Il est aussi possible de connaitre son type dans le cas d'une simulation avec plusieurs types d'agents.

5.2.3 Le système de proxy

Pour rappel dans le contexte des PDMAS, il est nécessaire de distribuer l'environnement sur plusieurs processus. De ce fait, les agents se déplacent d'un processus (AC) à un autre processus (ou AC) afin de garder la continuité de l'environnement et donc de pouvoir exécuter correctement leurs comportements. Si un agent souhaite envoyer un message à un autre agent, il doit connaitre sur quel processus l'agent qu'il souhaite contacter, est maintenant exécuté. Pour respecter la contrainte mono-thread, un système de proxy qui consiste à laisser une trace de chaque agent sur le processus qui à créé l'agent au début de la simulation est utilisé. Cette trace est ensuite mise à jour au cours de la simulation.

L'algorithme 3 détaille la mise à jour des proxys. Lorsqu'un agent se déplace d'un processus à un autre son proxy est mis à jour. Chaque AC contient une hashmap de proxys ayant une entrée pour chaque agent qu'il a créé au début de la simulation. Lorsqu'un agent est sur le point de se déplacer d'un processus à un autre, l'AC vérifie si cet agent a bien été créé par lui-même grâce à son identifiant (ligne 2). Si c'est le cas, l'AC met à jour la hashmap des proxys en modifiant la valeur du processus sur lequel l'agent s'exécute par la future valeur sur lequel l'agent va se déplacer (ligne 3). Dans les autres cas, c'est à dire si l'AC n'a pas créé l'agent au début de la simulation, alors il envoie un message à l'AC créateur de cet agent pour lui demander de mettre à jour sa table des proxys avec le processus sur lequel l'agent va se déplacer (ligne 5). Ensuite,

l'algorithme de mise à jour des proxy suit les mêmes règles de terminaison que pour la phase réception des messages. C'est à dire, qu'il est composé de deux parties, une partie exécuté par le coordinateur de la ligne 9 à la ligne 18 et l'autre partie exécuté par les autres processus de la ligne 20 à la ligne 35. En effet, il est de même que pour la phase de réception des messages, impossible de savoir combien de mises à jour sont à effectuer, ni à partir de combien de processus il est possible de recevoir des mises à jour. Pour cette raison, nous utilisons comme précédemment l'algorithme de terminaison basé sur l'anneau bi-directionnel (ligne 18 et lignes 28 à 35). Comme cette phase de mise à jour est effectuée avant l'exécution du nouveau pas de temps, les tables de proxys sont toujours à jour lors de l'exécution d'un pas de temps.

Algorithme 3 : Phase de mise à jour des proxys à chaque pas de temps par chaque AC

```
1  while (For all migrated agents) do
2      if (Agent.GetOwnerProc() = AC_ID) then
3          MapProxy[Agent.GID] ← AC_ID;
4      else
5          SendMsgMajProxy(Agent.GetOwnerProc(), Agent.FutProcess);
6          NbSendMigration++;
7      end if
8  end while
9  if (AC is coordinator) then
10     while ((NbSendUpdates ≠ NbAckReceived) && (nbEnd ≠ NbProcesses-1)) do
11         RecvMsg(Msg);
12         switch Msg.TAG do
13             case DATA_UPDATE_AGENT do MapProxy[Msg.GID] ← Msg.FutProcess;
                SendAck(Msg.Source);
14             case ACK_UPDATE do NbAckReceived++;
15             case END do nbEndReceived++;
16         end switch
17     end while
18     SendFinishUpdating(⌈NbProcesses/2⌉);
19     WaitForTermination();
20 else
21     while ((NbSendUpdates ≠ NbAckReceived) && (FinishUpdating ≠ false)) do
22         RecvMsg(Msg);
23         switch Msg.TAG do
24             case DATA_UPDATE_AGENT do MapProxy[Msg.GID] ← Msg.FutProcess;
                SendAck(Msg.Source);
25             case ACK_UPDATE do NbAckReceived++;
26             case NEXT do FinishUpdating ← true;
27         end switch
28     end while
29     SendEndToCoordinator(END);
30     if (AC.ID = ⌈NbProcesses/2⌉) then
31         SendFinishUpdating(PREVIOUS_AC);
32         SendFinishUpdating(NEXT_AC);
33     else if (AC.ID > ⌈NbProcesses/2⌉) then
34         SendFinishUpdating(NEXT_AC);
35     else
36         SendFinishUpdating(PREVIOUS_AC);
37     end if
38 end if
```

De cette manière, il est toujours possible de joindre un agent en envoyant un message au

processus qui a créé l'agent au début de la simulation. Si l'agent n'est pas exécuté sur le processus qui reçoit le message alors le processus transfert le message au processus qui peut alors délivrer le message à l'agent concerné (Figure 5.6). Le message sera alors acquitté par le bon processus. Comme chaque processus (ou AC) attend tous les acquittements des messages envoyés avant d'envoyer un message de fin (END) au coordinateur de la simulation, cela garantit que chaque message envoyé sera délivré au pas de temps $n+1$. De plus, le modélisateur n'a pas à se soucier de l'endroit où se situent les agents. La seule information obligatoire est de connaitre l'identifiant de l'agent que nous souhaitons contacter.

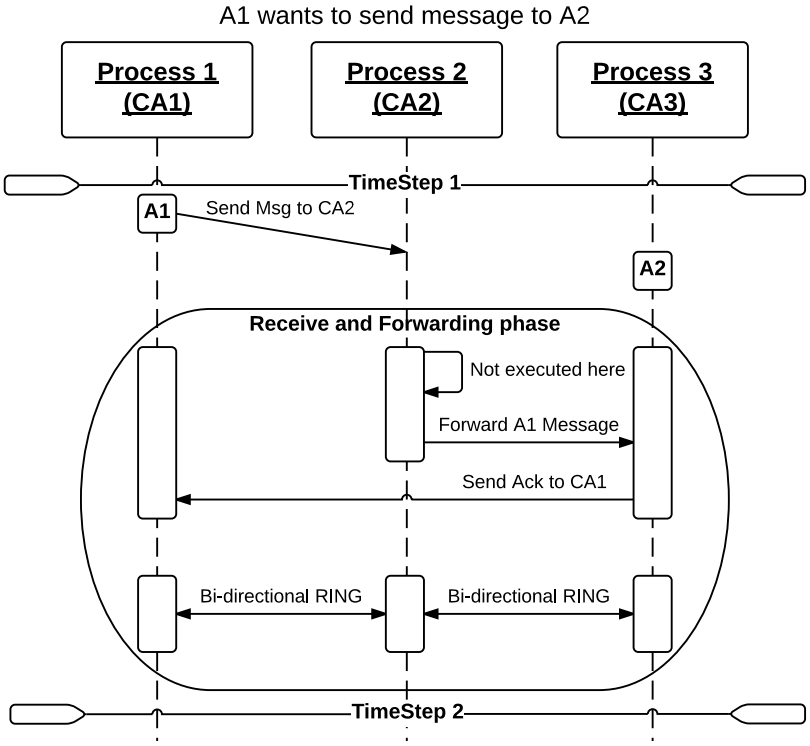


FIGURE 5.6 – Schéma de mise à jour exécuté à chaque pas de temps par chaque AC.

5.2.4 Les limites des communications distantes

Le schéma de communication proposé possède une limite qui est à prendre en compte lors du développement d'un modèle multi-agents. Comme énoncé précédemment, la simulation est exécutée par pas de temps, chaque pas de temps représentant un temps simulé dans la simulation. Avec le schéma de communication proposé, le nombre de communications qu'un agent peut effectuer avec un autre agent à chaque pas de temps est limité puisque chaque message envoyé au pas de temps n est reçu au pas de temps $n+1$. Ainsi, si le pas de temps simulé représente une grande durée de temps réel alors uniquement une communication (envoi ou réponse) entre deux agents identiques est possible dans ce pas de temps, ce qui ne reflète pas forcement la réalité, par exemple dans le cas d'une communication orale. Pour éviter cette limite, il faut donc un temps simulé très court pour que le schéma soit cohérent.

5.3 Expérimentations

Pour évaluer les performances de l'algorithme de communication, nous utilisons le modèle de référence définit au Chapitre 1 en section 1.1.5. Pour rappel, ce modèle de référence offre la possibilité d'évaluer uniquement les communications distantes entres les agents. C'est à dire, qu'au lieu d'envoyer des messages dans le champ de perception de l'agent, chaque agent envoie autant de messages à des agents aléatoirement choisis qui s'exécutent sur des autres processus qu'il y a d'agents dans son champ de perception. Le modèle de référence utilisé pour les expérimentations est basé sur une grille de 300x300 qui représente l'environnement où les personnes peuvent évoluer. A l'initialisation, 10000 personnes sont réparties de manière aléatoire dans l'environnement. La Table 5.1 présente les paramètres d'expérimentation utilisés. Il est important de noter que pour des soucis de reproductibilité, la même configuration initiale est utilisée pour toutes les exécutions des simulations, seule la graine varie d'une exécution à une autre. Le champ de perception de chaque agent est fixé à 3 tandis que la taille du calcul de DFT est fixées à 128. Les simulations sont exécutées durant 200 pas de temps. La Table 5.1 présente les jeux de paramètres détaillés pour les expérimentations du modèle de référence.

Paramètres	Jeu de valeurs 1
Environnement	300x300
Nb. Agent	10000
Chp. perception	3
Taille DFT	128
Nb. pas de temps	200

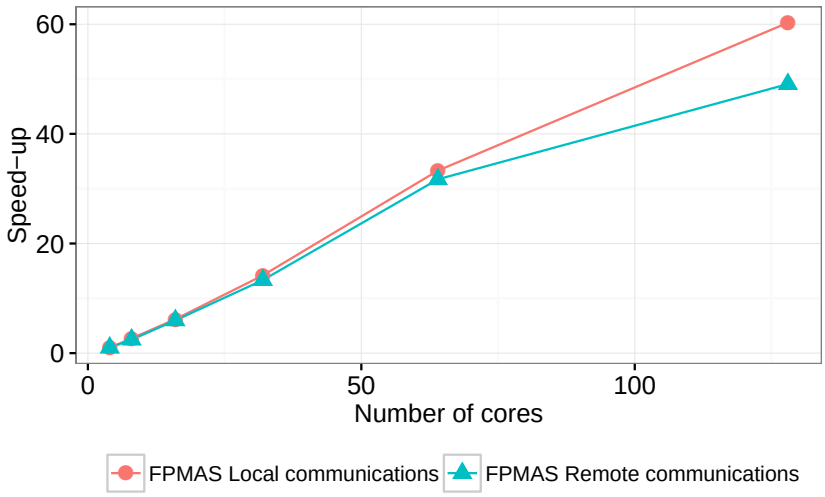
TABLE 5.1 – Jeux de paramètres pour les expérimentations du modèle de référence

La Figure 5.7 représente l'extensibilité 5.7(a) et les temps d'exécutions 5.7(b) de la plateforme FPMAS pour des communications locales et distantes. Pour calculer l'extensibilité nous faisons varier le nombre de cœurs utilisés pour exécuter la même simulation tandis que nous fixons le nombre d'agents.

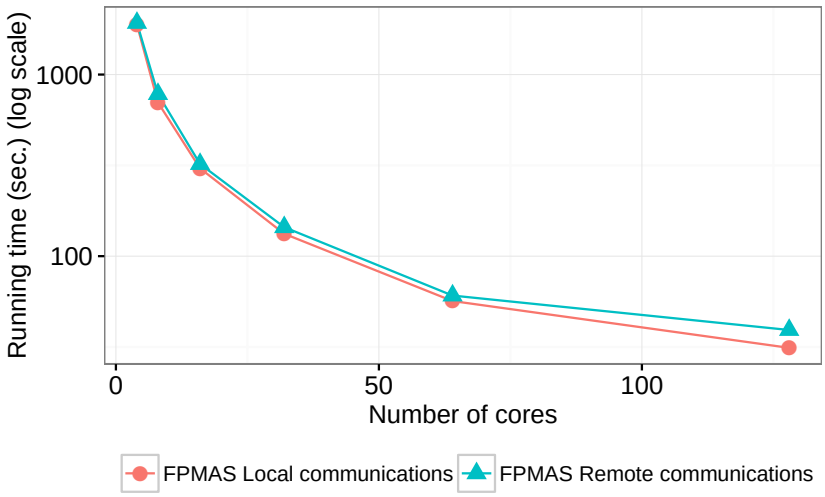
Comme nous pouvons le constater, les deux schémas de communication, sont extensibles, même si celui avec uniquement les communications locales possède une bien meilleure extensibilité. La différence entre les deux speed-up est assez marquée, environ 18% de différence pour 128 cœurs. Évidemment, le schéma permettant les communications distantes offre des performances moins élevées, ce qui s'explique simplement par la quantité plus importante de messages échangés entre les processus. En effet, plus il y aura d'interactions distantes, plus cela prendra du temps.

La Figure 5.8 représente quant à elle la monté en charge de notre schéma de communication. Pour cela nous fixons le nombre de cœurs à 8 et nous faisons varier le nombre d'agents de 10000 à 30000, ensuite nous comparons le temps d'exécution des communications uniquement locales avec celui des communications distantes.

De la même manière que pour les résultats d'extensibilité, les résultats d'exécution des communications distantes, ne peuvent être meilleurs que les résultats des communications locales. Cependant, les résultats d'exécution avec des communications distantes montrent que la charge est plutôt bien supportée. Par exemple, pour 5000 agents, il y a seulement une différence d'environ 1.5% entre les exécutions de communications locales et distantes et pour 30000 agents, où la différence est la plus grande, elle est d'environ 5%. Ces valeurs peuvent être considérées comme acceptables suivant les modèles. Soulignons que des améliorations peuvent être effectuée autant sur le schéma de communication que sur l'implémentation de la plateforme qui n'est qu'une preuve de concept sans optimisation.



(a) Speed-up



(b) Running time

FIGURE 5.7 – Extensibilité et temps d'exécutions pour la plateforme FractalPMAS avec le modèle de référence de 4 à 128 cœurs pour 200 pas de temps

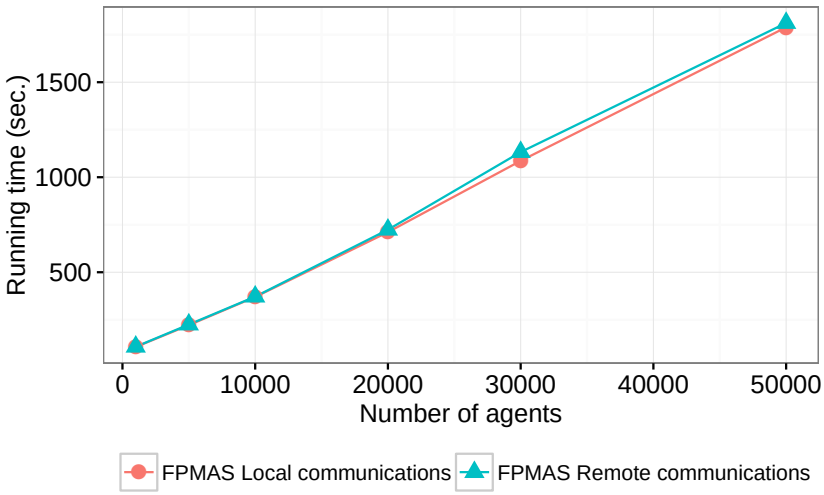


FIGURE 5.8 – Montée en charge pour la plateforme FractalPMAS avec le modèle de référence de 1000 à 30000 agents utilisant 8 cœurs et 200 pas de temps

Conclusion et Perspectives

Ce dernier chapitre, dresse un bilan du travail effectué durant cette thèse. Nous présentons les réponses que nous avons apportées avec ces travaux dans le domaine des systèmes multi-agents parallèles. Ensuite, nous proposons quelques perspectives afin de continuer ces travaux mais aussi d'enrichir les résultats obtenus.

Analyse et bilan

La principale question posée par ce travail de thèse est de savoir s'il est possible d'élaborer ou d'utiliser des techniques ou méthodes issues du domaine du parallélisme pour permettre aux systèmes multi-agents d'être parallélisés de manière efficace afin de bénéficier de plus de puissance de calcul et de mémoire. Pour ce faire, nous avons tenté de répondre aux différentes questions posées par la problématique présentée en introduction de ce mémoire :

— **Q1** : Quelle modélisation utiliser pour efficacement distribuer une simulation multi-agents au sein d'un environnement parallèle ?

— **Q2** : Comment synchroniser et garantir la cohérence de la simulation multi-agents parallèle ?

— **Q3** : Comment gérer la communication entre les agents dans une simulation multi-agents parallèle ?

La réponse à ces questions requiert une connaissance des plateformes multi-agents parallèles afin de dresser un constat de l'existant. Pour cela, nous avons effectué un comparatif détaillé des plateformes multi-agents parallèles à deux niveaux : qualitatif et quantitatif, c'est-à-dire une analyse bibliographique et une évaluation des performances sur un super-calculateur en passant par l'implémentation de modèles dans chacune des plateformes. Ce comparatif a montré que des limites existaient dans les plateformes actuelles. Elles sont les suivantes :

— Des problèmes de distribution de la simulation qui impliquent une manque de flexibilité et impactent les performances.

— Des problèmes de synchronisation car la plupart des plateformes ne gère pas tous les cas de synchronisation, et en particulier les écritures concurrentes.

— Des problèmes de communications entre agents ne permettant pas pour toutes les plateformes de prendre en charge efficacement tous les cas de communications possibles.

Pour répondre à ces questions de recherches nous avons proposé différentes contributions que nous détaillons dans la section suivante.

Solutions proposées

Le formalisme de *Nested Graphs*, utilisé pour modéliser les simulations multi-agents, offre un moyen graphique et intuitif de représenter les comportements des agents mais aussi l'environnement dans lequel les agents évoluent. Grâce à la modélisation multi-niveaux inhérente aux

Nested Graphs et à la structure de graphe, nous pouvons utiliser des outils de partitionnements existants. Ces outils de partitionnements, permettent entre autre de distribuer efficacement la simulation sur plusieurs processus mais aussi d'équilibrer dynamiquement la charge entre eux en se basant sur la densité d'agents présents. Cette distribution offre un grain plus fin que la décomposition cartésienne uniquement basée sur des divisions rectilignes. Ces propositions, nous permettent de répondre à la question de recherche **Q1**.

L'analyse de l'impact de la synchronisation sur les résultats de simulations et d'exécutions, est un travail qui vise à sensibiliser la communauté agents aux problèmes liés à la parallélisation d'un modèle agent. En effet, à travers nos expérimentations, nous quantifions le coût de la synchronisation dans la simulation suivant les modèles qui sont animés. Ce coût de synchronisation, non négligeable, est important à prendre en compte lors de la modélisation d'un modèle en parallèle. Suivant le modèle que l'on souhaite implémenter, la synchronisation peut être adaptée afin d'obtenir les meilleurs performances lors de l'exécution. C'est la raison pour laquelle, nous avons défini différentes politiques de synchronisation qui permettent de gérer tous les types de modèles. La synchronisation est, de fait, dépendante des comportements des agents. C'est à la charge du modélisateur de prêter attention à la manière dont s'exécutera le modèle et donc d'adapter la modélisation au support proposé par la plateforme. Cette analyse, nous permet de répondre à la question de recherche **Q2**.

L'un des principes de base des agents est l'interaction et la communication. La distribution de la simulation agents sur plusieurs processus implique que chaque agent et processus n'a qu'une connaissance partielle de l'endroit où se situent les autres agents. De ce fait, cela rend difficile la communication entre les agents ne s'exécutant pas sur le même processus. Cependant, de nombreux modèles agents, en particulier, les modèles de mobilité urbaine nécessitent de communiquer avec l'ensemble des agents. Pour cela nous avons proposé des algorithmes permettant à tous les agents de la simulation de communiquer sans porter attention aux processus où ils s'exécutent. Ces algorithmes reposent sur une identification particulière des agents qui permet de connaître à tout moment où un agent s'exécute. Le schéma que nous proposons garantit la réception de tous les messages envoyés. Pour finir, ce schéma de communication est basé sur le standard de communication MPI et peut être utilisé comme support pour implémenter des standards de communication agent comme FIPA. Ce schéma de communication, nous permet de répondre à la question de recherche **Q3**.

Expérimentations

Les expérimentations menées au cours de cette thèse nous ont permis dans un premier temps de tester et d'évaluer les plateformes multi-agents parallèles existantes et fonctionnelles sur un super-calculateur. Ces premières expérimentations, nous ont permis d'observer le comportement de ces plateformes. Dans un deuxième temps, les expérimentations, nous ont permis d'étudier et de valider nos propositions à l'aide de la plateforme implémentée FractalPMAS.

Ces expérimentations représentent environ 352 000 heures de calculs sur le Mésocentre de calcul de Franche-comté [1].

Perspectives

Au vue des résultats encourageants obtenus au cours de cette thèse, nous présentons dans cette section différentes perspectives possibles à ces travaux. Les deux premières perspectives ou améliorations concernent la plateforme FractalPMAS tandis que les deux dernières perspectives ouvrent sur de nouvelles problématiques.

1. `http://meso.univ-fcomte.fr/`

Amélioration de la plateforme FractalPMAS

A ce jour, la plateforme FractalPMAS est en partie fonctionnelle mais reste à l'état de preuve de concept. La plateforme permet d'implémenter un modèle multi-agents à l'aide du formalisme de *Nested Graphs* puis de l'exécuter dans un environnement parallèle de type super-calculateur HPC. De nombreuses améliorations doivent être apportées à la plateforme FractalPMAS pour qu'elle soit complète et distribuable.

Par exemple, la bibliothèque CGraph qui permet d'implémenter la structure de *Nested Graphs* sur laquelle repose la plateforme est uniquement implémentée à l'aide de tableaux. La recherche d'un agent n'est donc pas optimisée car il faut parcourir, dans le pire des cas, la totalité du tableau. Une piste d'amélioration serait de proposer trois implémentations de la bibliothèque CGraph : une à l'aide de tableaux, une à l'aide de tables de hachage et une à l'aide d'ensembles ordonnés. De cette manière, le modélisateur pourrait, suivant le modèle à implémenter, choisir la structure la plus adaptée. De ce fait, la recherche d'un agent pourrait être améliorée et offrir davantage de performances pour la plateforme FPMAS.

Génération automatique de code parallèle

Pour faciliter l'implémentation d'un modèle, un générateur de code pourrait être mis en place pour générer le code des agents à partir de la sémantique graphique des comportements des agents à l'aide des *Nested Graphs*. En effet, un outil de modélisation graphique permettrait d'offrir la possibilité à des modélisateurs n'ayant pas toutes les compétences techniques pour implémenter un modèle directement en langage C++, d'utiliser la plateforme FractalPMAS.

Application des *Nested Graphs* à la simulation numérique

La structure de *Nested Graphs* permet la représentation de n'importe quel élément. Ainsi, il est possible d'appliquer non seulement les *Nested Graphs* aux systèmes multi-agents parallèles mais à la simulation numérique de manière générale. En effet, la structure de *Nested Graphs* permettrait d'effectuer des groupements de calculs à effectuer sur plusieurs niveaux et ainsi de paralléliser davantage en utilisant des architectures hybrides tel que : GPGPU ou XeonPhi.

Validation et vérification des systèmes multi-agents

Un des avantages de la structure de *Nested Graphs*, en plus d'être basée sur un langage graphique, est qu'elle repose sur une description formelle et mathématique. Les algorithmes classiques de graphes peuvent donc être appliqués. Grâce à la sémantique formelle des *Nested Graphs*, il serait intéressant d'utiliser des bibliothèques de méthodes formelles [95] pour essayer de valider et vérifier les modèles multi-agents parallèles. Par exemple, cela permettrait de garantir qu'aucune situation d'inter-blocage n'est présente dans le modèle.

Bibliographie

[1] Michael Wooldridge and Nicholas R Jennings. Intelligent agents : Theory and practice. *The knowledge engineering review*, 10(02) :115–152, 1995.

[2] Nick Collier. Repast hpc manual, 2010.

[3] Xavier Rubio-Campillo. Pandora : A versatile agent-based modelling platform for social simulation. In *SIMUL 2014, The Sixth International Conference on Advances in System Simulation*, pages 29–34, Nice, France, 2014. IARIA.

[4] Xavier Rubio-Campillo and José Ma. Cela. Large-scale agent-based simulation in archaeology : an approach using high-performance computing. In *BAR International Series 2494. Proceedings of the 38th Annual Conference on Computer Applications and Quantitative Methods in Archaeology, Granada, Spain, April 2010*, pages 153–159, 2013 2013.

[5] Au LS Chin, Au DJ Worth, Au C Greenough, Au S Coakley, M Holcombe, and M Kiran. Flame : An approach to the parallelisation of agent-based applications. *Work*, 501 :63259, 2012.

[6] Karen D Devine, Erik G Boman, Lee Ann Riesen, Umit V Catalyurek, and Cédric Chevalier. Getting started with zoltan : A short tutorial. *Sandia National Labs Tech Report SAND2009-0578C*, 2009.

[7] James Ladyman, James Lambert, and Karoline Wiesner. What is a complex system ? *European Journal for Philosophy of Science*, 3(1) :33–67, 2013.

[8] Gregory Carslaw. *Agent based modelling in social psychology.* PhD thesis, University of Birmingham, 2013.

[9] Eliot R Smith and Frederica R Conrey. Agent-based modeling : A new approach for theory building in social psychology. *Personality and social psychology review*, 11(1) :87–104, 2007.

[10] Jacques Ferber and Jean-François Perrot. *Les systèmes multi-agents : vers une intelligence collective.* InterEditions Paris, 1995.

[11] Anand S Rao, Michael P Georgeff, et al. Bdi agents : From theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[12] Brahim Chaib-Draa, Imed Jarras, and Bernard Moulin. Systèmes multi-agents : principes généraux et applications. *Edition Hermès*, 2001.

[13] Jean-Pierre Müller. *Des systèmes autonomes aux systèmes multi-agents : Interaction, émergence et systèmes complexes.* PhD thesis, Université Libre de Bruxelles, 2002.

[14] R Menzel, M Hammer, U Müller, and H Rosenboom. Behavioral, neural and cellular components underlying olfactory learning in the honeybee. *Journal of Physiology-Paris*, 90(5) :395–398, 1996.

[15] Michael Wooldridge. On the sources of complexity in agent design. *Applied Artificial Intelligence*, 14(7) :623–644, 2000.

[16] Jürgen Lind. *Iterative software engineering for multiagent systems : the MASSIVE method*. Springer-Verlag, 2001.

[17] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence : a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[18] Lars Braubach, Alexander Pokahr, Winfried Lamersdorf, Karl-Heinz Krempels, and Peer-Oliver Woelk. A generic time management service for distributed multi-agent systems. *Applied Artificial Intelligence*, 20(2-4) :229–249, 2006.

[19] Richard M Fujimoto. Parallel simulation : distributed simulation systems. In *Proceedings of the 35th Conf. on Winter simulation : driving innovation*, pages 124–134. Winter Simulation Conf., 2003.

[20] Arnaud Banos, Christophe Lang, and Nicolas Marilleau. *Simulation spatiale à base d'agents avec NetLogo 1 : introduction et bases*. 2015.

[21] Olivier Gutknecht and Jacques Ferber. Madkit : A generic multi-agent platform. In *Proceedings of the fourth international Conf. on Autonomous agents*, pages 78–79. ACM, 2000.

[22] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, and Keith Sullivan. MASON : A New Multi-Agent Simulation Toolkit. *Simulation*, 81(7) :517–527, July 2005.

[23] François Bousquet, Innocent Bakam, Hubert Proton, and Christophe Le Page. Cormas : common-pool resources and multi-agent systems. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 826–837. Springer, 1998.

[24] Nick Collier. Repast : An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36 :2003, 2003.

[25] Uri Wilensky. {NetLogo}. 1999.

[26] Edouard Amouroux, Thanh-Quang Chu, Alain Boucher, and Alexis Drogoul. Gama : an environment for implementing and running spatially explicit multi-agent simulations. In *Agent computing and multi-agent systems*, volume 5044, pages 359–371. Springer, 2009.

[27] Corey McCaffrey. Starlogo tng : the convergence of graphical programming and text processing. *Massachusetts Institute of Technology Master's Thesis*, 2006.

[28] Patrice Langlois, Baptiste Blanpain, and Eric Daudé. Magéo, une plateforme de simulation multi-agents pour tous. In *SimTools 2013*, 2013.

[29] Robert Tobias and Carole Hofmann. Evaluation of free java-libraries for social-scientific agent based simulation. *JASS*, 7(1), 2004.

[30] Rafael H Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J Gomez-Sanz, Joao Leite, Gregory MP O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1) :33–44, 2006.

[31] Matthew Berryman. Review of software platforms for agent based models. Technical report, DTIC Document, 2008.

[32] Brian Heath, Raymond Hill, and Frank Ciarallo. A survey of agent-based modeling practices (january 1998 to july 2008). *JASSS*, 12(4) :9, 2009.

[33] Uri Wilensky. Netlogo flocking model. *Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL*, 1998.

[34] Craig W Reynolds. Flocks, herds and schools : A distributed behavioral model. *ACM SIGGRAPH computer graphics*, 21(4) :25–34, 1987.

[35] U Wilensky. Netlogo wolf sheep predation model. evanston, il : Center for connected learning and computer-based modeling, northwestern university, 1998.

[36] JAMES A Yorke, NEAL Nathanson, GIULIO Pianigiani, and JOHN Martin. Seasonality and the requirements for perpetuation and eradication of viruses in populations. *American Journal of Epidemiology*, 109(2) :103–123, 1979.

[37] U Wilensky. Netlogo wolf sheep predation (docked) model. *Online] Center for connected learning and computer-based modeling, Northwestern University, Evanston, IL. Available at : http ://ccl. northwestern. edu/netlogo/models/WolfSheepPredation (docked).[Accessed 2 June 2009]*, 2005.

[38] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2) :216–231, 2005.

[39] Leonardo Dagum and Ramesh Menon. Openmp : an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1) :46–55, 1998.

[40] Mark Baker. Cluster computing white paper. *arXiv preprint cs/0004014*, 2000.

[41] The MPI Forum. Mpi : A message passing interface, 1993.

[42] Randal Everitt Bryant. Simulation of packet communication architecture computer systems. 1977.

[43] K. Mani Chandy and Jayadev Misra. Distributed simulation : A case study in design and verification of distributed programs. *IEEE Transactions on software engineering*, (5) :440–452, 1979.

[44] Behrokh Samadi. Distributed simulation, algorithms and performance analysis (load balancing, distributed processing). 1985.

[45] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3) :404–425, 1985.

[46] Guillaume Laville, Christophe Lang, Bénédicte Herrmann, Laurent Philippe, Kamel Mazouzi, and Nicolas Marilleau. Mcmas : A toolkit for developing agent-based simulations on many-core architectures. *Multiagent and Grid Systems*, 11(1) :15–31, 2015.

[47] Stéphane Vialle, Eugen Dedu, and Claude Timsit. Parcel-5/parssap : A parallel programming model and library for easy development and fast execution of simulations of situated multi-agent systems. In *Proceedings of SNPD'02 International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing*, 2002.

[48] N Bezirgiannis. Improving performance of simulation software using haskell's concurrency & parallelism. Master's thesis, Utrecht University, 2013.

[49] Franco Cicirelli, Andrea Giordano, and Libero Nigro. Efficient environment management for distributed simulation of large-scale situated multi-agent systems. *Concurrency and Computation : Practice and Experience*, 27(3) :610–632, 2015.

[50] Michael Lees, Brian Logan, Rob Minson, Ton Oguara, and Georgios Theodoropoulos. *Modelling Environments for Distributed Simulation*, pages 150–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[51] Simon Coakley, Marian Gheorghe, Mike Holcombe, Shawn Chin, David Worth, and Chris Greenough. Exploitation of hpc in the flame agent-based simulation framework. In *Proceedings of the 2012 IEEE 14th Int. Conf. on HPC and Communication & 2012 IEEE 9th Int. Conf. on Embedded Software and Systems*, HPCC '12, pages 538–545, Washington, DC, USA, 2012. IEEE Computer Society.

[52] Nicholson Collier and Michael North. *Repast HPC : A platform for large-scale agentbased modeling*. Wiley, 2011.

[53] Gennaro Cordasco, Rosario Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. A Framework for Distributing Agent-Based Simulations. In *Euro-Par 2011 : Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 460–470, 2011.

[54] Elaini S Angelotti, Edson E Scalabrin, and Bráulio C Ávila. Pandora : a multi-agent system using paraconsistent logic. In *Computational Intelligence and Multimedia Applications, 2001. ICCIMA 2001.*, pages 352–356. IEEE, 2001.

[55] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[56] Ton Oguara, G Theodoropoulos, B Logan, M Lees, and C Dan. PDES-MAS : A unifying framework for the distributed simulation of multi-agent systems. Technical Report CSR-07-7, School of computer science research, University of Birmingham, july 2007.

[57] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. Pdes-mas : Distributed simulation of multi-agent systems. *Procedia Comp. Sc.*, 18 :671–681, 2013.

[58] M Scheutz, P Schermerhorn, R Connaughton, and A Dingler. Swages-an extendable distributed experimentation system for large-scale agent-based alife simulations. *Proceedings of Artificial Life X*, pages 412–419, 2006.

[59] Russell K Standish and Richard Leow. Ecolab : Agent based modeling for c++ programmers. *arXiv preprint cs/0401026*, 2004.

[60] Les Gasser and Kelvin Kakugawa. Mace3j : fast flexible distributed simulation of large, large-grain multi-agent systems. In *Proceedings of the first inter. joint Conf. on Autonomous agents and multiagent systems : part 2*, pages 745–752. ACM, 2002.

[61] Simon Coakley, Rod Smallwood, and Mike Holcombe. Using x-machines as a formal basis for describing agents in agent-based modelling. *SIMULATION SERIES*, 38(2) :33, 2006.

[62] Anthony J Cowling, Horia Georgescu, Marian Gheorghe, Mike Holcombe, and Cristina Vertan. Communicating stream x-machines systems are no more than x-machines. *Journal of Universal Computer Science*, 5(9) :494–507, 1999.

[63] Mike Holcombe, Simon Coakley, and Rod Smallwood. A general framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European conference on complex systems*. European Complex Systems Society Paris, France, 2006.

[64] Franco Cicirelli and Libero Nigro. Control centric framework for model continuity in time-dependent multi-agent systems. *Concurrency and Computation : Practice and Experience*, 2016.

[65] Makoto Matsumoto and Takuji Nishimura. Mersenne twister : a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1) :3–30, 1998.

[66] George Marsaglia and Arif Zaman. A new class of random number generators. *The Annals of Applied Probability*, pages 462–480, 1991.

[67] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. *Automatic nonuniform random variate generation*. Springer, 2004.

[68] Claudio Márquez, Eduardo César, and Joan Sorribes. A load balancing schema for agent-based spmd applications. In *International Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2013.

[69] Jan Himmelspach and Adelinde M. Uhrmacher. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium*, ANSS '07, pages 137–143, Washington, DC, USA, 2007. IEEE Computer Society.

[70] Udo Inden Paulo Leitão and Claus-Peter Rückemann. Parallelising multi-agent systems for high performance computing. In *INFOCOMP 2013 : The Third International Conference on Advanced Communications and Computation*, Lisbon, Portugal, 2013. IARIA.

[71] D. Pawlaszczyk and S. Strassburger. Scalability in distributed simulations of agent-based models. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1189–1200, Dec 2009.

[72] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis : Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[73] Kalyan S Perumalla. μsik-a micro-kernel for parallel/distributed simulation systems. In *Principles of Advanced and Distributed Simulation, 2005. PADS 2005. Workshop on*, pages 59–68. IEEE, 2005.

[74] Gennaro Cordasco, Francesco Milone, Carmine Spagnuolo, and Luca Vicidomini. Exploiting d-mason on parallel platforms : A novel communication strategy. In *Euro-Par Workshops (1)'14*, pages 407–417, 2014.

[75] Peter Wittek and Xavier Rubio-Campillo. Scalable agent-based modelling with cloud hpc resources for social simulations. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conf. on*, pages 355–362. IEEE, 2012.

[76] Biagio Cosenza, Gennaro Cordasco, Rosario De Chiara, and Vittorio Scarano. Distributed load balancing for parallel agent-based simulations. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 62–69. IEEE, 2011.

[77] Marco Madella, Bernardo Rondelli, Carla Lancelotti, Andrea Balbo, Debora Zurro, Xavi Rubio Campillo, and Sebastian Stride. Introduction to simulating the past. *Journal of Archaeological Method and Theory*, 21(2) :251–257, 2014.

[78] Simon Coakley, Paul Richmond, Marian Gheorghe, Shawn Chin, David Worth, Mike Holcombe, and Chris Greenough. Large-scale simulations with flame. *Intelligent Agents in Data-intensive Computing*, 14 :123, 2015.

[79] Michele Carillo, Gennaro Cordasco, Rosario De Chiara, Francesco Raia, Vittorio Scarano, and Flavio Serrapica. Enhancing the performances of d-mason - a motivating example. In Nuno Pina, Janusz Kacprzyk, and Mohammad S. Obaidat, editors, *SIMULTECH*, pages 137–143. SciTePress, 2012.

[80] Mehmet Can Kurt, Sriram Krishnamoorthy, Kunal Agrawal, and Gagan Agrawal. Fault-tolerant dynamic task graph scheduling. In *High Performance Computing, Networking, Storage and Analysis, SC14 : International Conference for*, pages 719–730. IEEE, 2014.

[81] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12) :1519–1534, 2000.

[82] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version*, 2, 2003.

[83] Cédric Chevalier and François Pellegrini. Pt-scotch : A tool for efficient parallel graph ordering. *CoRR*, abs/0907.1375, 2009.

[84] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2) :90–96, 2002.

[85] Sivasankaran Rajamanickam and Erik G Boman. An evaluation of the zoltan parallel graph and hypergraph partitioners. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2012.

[86] Patrick Taillandier, Duc-An Vo, Edouard Amouroux, and Alexis Drogoul. GAMA : A Simulation Platform That Integrates Geographical Information Data, Agent-Based Modeling and Multi-scale Control. In Nirmit Desai, Alan Liu, and Michael Winikoff, editors, *Principles and Practice of Multi-Agent Systems*, volume 7057, pages 242–258. Springer Berlin Heidelberg, 2012.

[87] Alexandra Poulovassilis and Mark Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems (TOIS)*, 12(1) :35–68, 1994.

[88] Sébastien Picault and Philippe Mathieu. An interaction-oriented model for multi-scale simulation. In *IJCAI'2011–Barcelona (Spain)–July, 16-22 2011*, pages 332–337. AAAI Press, 2011.

[89] Jose R Celaya, Alan Desrochers, Robert J Graves, et al. Modeling and analysis of multi-agent systems using petri nets. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 1439–1444. IEEE, 2007.

[90] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing : Partitioning, ordering, and coloring. *Scientific Programming*, 20(2) :129–150, 2012.

[91] Karen D Devine, Erik G Boman, Robert T Heaphy, Umit V Catalyürek, and Robert H Bisseling. Parallel hypergraph partitioning for irregular problems. *SIAM Parallel Processing for Scientific Computing*, 2006.

[92] C Greenough GL Poulter. *FLAME Tutorial Examples : Predation - a simple predator-prey model*. STFC, 2014.

[93] John T Murphy. Computational social science and high performance computing : A case study of a simple model at large scales. In *Proceedings of the 2011 Annual Conference of the Computational Social Science Society of America*, 2011.

[94] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[95] Edmund M Clarke and Jeannette M Wing. Formal methods : State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4) :626–643, 1996.

Document réalisé avec LaTeX et :
le style LaTeX pour Thèse de Doctorat créé par S. Galland — http://www.multiagent.fr/ThesisStyle
la collection de paquets tex-upmethodology — http://www.arakhne.org/tex-upmethodology/

Résumé :

Les travaux de cette thèse s'inscrivent dans le domaine des systèmes complexes et s'intéressent plus particulièrement à l'exécution efficace et reproductible de simulations multi-agents de grande taille dans un contexte parallèle et distribué de haute performance de type cluster (HPC). Dans ce contexte, nous nous intéressons plus particulièrement à la conception des modèles pour faciliter leur distribution, à la synchronisation des composants distribués et à la communication entre agents. La première contribution de cette thèse est la comparaison qualitative et quantitative des principales plateformes multi-agents parallèles et distribués qui ciblent les simulations à large échelle dans un environnement haute performance. Ce travail a permis d'identifier les limites ou manques des plateformes existantes, majoritairement la communication entre les agents, la synchronisation ainsi que la distribution de la charge peu flexible. Pour offrir plus de flexibilité à la distribution des simulations, nous proposons un formalisme de modélisation à base de graphes imbriqués qui nous permet de tirer parti de librairies performantes pour décomposer et distribuer les simulations. Nous avons ensuite effectué une étude sur l'impact de la synchronisation dans les PDMAS, en proposant trois politiques de synchronisation différentes afin de fournir aux modélisateurs un niveau de résolution adapté aux différents problèmes de synchronisation. Pour finir, nous définissons un schéma de communication entre toutes les entités qui composent une simulation indépendamment du processus sur lequel les entités s'exécutent.

Ces propositions sont réunies au sein d'une plateforme multi-agents parallèle appelée FractalPMAS. Cette plateforme est une preuve de concept qui nous a permis de mettre en œuvre nos différentes contributions afin d'observer et de comparer les comportements de nos algorithmes. Pour valider ce travail trois modèles agents reconnus, le modèle proie-prédateur, le modèle Flocking et un modèle de contamination, ont été utilisés. Nous avons réalisé des simulations utilisant jusqu'à 512 cœurs et les résultats obtenus, en termes de performances et d'extensibilité, s'avèrent prometteurs.

Mots-clés : Simulation, Multi-agent, Parallélisme, Distribution, Nested Graph

Abstract:

Contributions of this PHD take place on computer science research on complex systems, specifically in efficient and reproducible execution of large multi-agent simulations in a parallel and distributed high performance cluster type of context (HPC) systems. We are particularly interested in the design of models to facilitate their distribution, synchronization of distributed components and communication between agents. In this context, we are particularly interested in the design of models to facilitate their distribution, synchronization of distributed components and communication between agents. The first contribution of this thesis is the qualitative and quantitative comparison of the main parallel and distributed multi-agent platforms targeting large scale simulations in a high performance environment. This work identified limitations of existing platforms, mainly communication between agents, synchronization and the distribution of the load which is inflexible. To offer more flexibility in the distribution of simulations, we propose a modeling formalism based nested graphs allowing us to decompose and distributed simulations using powerful libraries. We then conducted a study on the impact of synchronization in PDMAS, proposing three different synchronization policies to provide modelers a level of resolution adapted to the various synchronization problems. Finally, we define a communication schema between all entities that make up a simulation regardless of the process on which entities are running.

These contributions are combined in a parallel multi-agent platform called FractalPMAS. This platform is a proof of concept and allowed us to implement our different contributions to observe and compare the behavior of our algorithms. To validate this work three recognized agents model, the predator-prey model, the Flocking model and contamination model were used. We performed simulations using up to 512 cores and the results obtained, in terms of performance and scalability are promising.

Keywords: Simulation, Multi-Agent, Parallelism, Distribution, Nested Graph

SPIM

■ École doctorale SPIM 1 rue Claude Goudimel F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ **ed-spim@univ-fcomte.fr** ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ
DE FRANCHE-COMTÉ