Distribution et synchronisation des simulations de Systèmes Multi-Agents

Paul Breugnot

THÈSE DE DOCTORAT DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

École doctorale n°37

Sciences Physiques pour l'Ingénieur et Microtechniques (SPIM)

Doctorat d'informatique

Par

Paul BREUGNOT

Distribution et synchronisation des simulations de Systèmes Multi-Agents

Thèse présentée et soutenue à Besançon, le 16 mars 2023

Composition du Jury :

Chevrier, Vincent	Professeur, Université de Lorraine	Président
Picard, Gauthier	Directeur de recherche, ONERA	Rapporteur
Drogoul, Alexis	Directeur de recherche, Institut de Recherche et Développement	Rapporteur
Hill, David	Professeur, Université de Clermont-Auvergne	Examinateur
Philippe, Laurent	Professeur, Université de Franche-Comté	Directeur de thèse
Herrmann, Bénédicte	Maître de conférence, Université de Franche-Comté	Encadrant de thèse
Lang, Christophe	Maître de conférence HDR, Université de Franche-Comté	Encadrant de thèse
Marilleau, Nicolas	Ingénieur de recherche HDR, Institut de Recherche et Développement	Invité

Remerciements

De nature assez peu confiante, j'ai eu tout le temps de ressasser l'utilité et la légitimité de mes travaux au cours des trois années passées. De nombreuses personnes ont pourtant permis l'aboutissement de ce projet, dont je suis aujourd'hui très fier.

J'ai en effet pu bénéficier d'un encadrement d'une qualité exceptionnelle, que je souhaite à tout futur doctorant. Ainsi je remercie d'abord Bénédicte Herrmann, pour tout le temps passé à faire des lectures éclairées de nos articles ou de ce manuscrit. Son analyse pertinente des problèmes abordés en termes de Systèmes Multi-Agents comme en parallélisme et sa ténacité face à mon pessimisme récurrent m'ont permis de comprendre et de valoriser l'essentiel de nos résultats. Sa clairvoyance nous aura évité bien des égarements, en réunion comme en rédaction.

Je remercie ensuite Christophe Lang, dont la passion pour les Systèmes Multi-Agents s'avère communicative. C'est un plaisir et un privilège d'avoir pu bénéficier de son expérience dans le domaine. Ses compétences dans la gestion des problèmes administratifs ainsi que sa bonne humeur en toute circonstance n'ont par ailleurs cessé de m'impressionner, tout comme la qualité et l'audace de ses jeux de mots, qui bien souvent ne sont compris facilement que par son binôme de longue date en la matière, Laurent Philippe.

Je remercie ce dernier pour son écoute et ses précieux conseils techniques en termes de calcul haute performance et de randonnée pédestre. En trois ans, je n'ai pu cerner qu'une infime portion des problèmes complexes du parallélisme qui aujourd'hui ne sont pour lui que des formalités. Sa détermination à résoudre les problèmes me donne bon espoir en l'avenir de la simulation distribuée de Systèmes Multi-Agents. Son travail pour le Mésocentre fut également un atout indéniable.

Je remercie dans ce contexte Kamel Mazouzi, Sékou Diakité et toute l'équipe du Mésocentre pour leur réactivité et leur gestion exemplaire du cluster, qui n'a jamais failli pendant nos milliers d'heures de calcul.

Les cours de Systèmes Multi-Agents dispensés par Gauthier Picard ont certainement participé à mon engagement dans ce domaine. Je le remercie, ainsi qu'Alexis Drogoul, pour leur travail en tant que rapporteurs de cette thèse.

Je remercie aussi les étudiants qui m'ont fait changer d'air, tous les amis, collègues du DISC et camarades de Besançon ou d'ailleurs mais jamais vraiment loin, ainsi que ma famille du Morvan chez qui j'ai toujours trouvé refuge.

Merci enfin de tout mon cœur à Louna, sans qui la vie serait bien triste. Les moments passés ensemble m'ont permis d'avancer et d'évoluer en toute circonstance, et je n'ose imaginer comment j'aurais pu traverser cette période de ma vie sans elle.

Table des matières

Première partie

Systèmes Multi-Agents et simulation distribuée

Cette thèse, réalisée au département DISC du laboratoire FEMTO-ST, fait état de mes travaux dans le cadre de la simulation distribuée de Systèmes Multi-Agents (SMA). Elle fait notamment suite à la thèse d'Alban Rousset [111] intitulée « Contribution à la distribution et à la synchronisation des Systèmes Multi-Agents sur les super-calculateurs », dont les avancées incluent une revue détaillée des plateformes de simulation distribuée de SMA existantes, la proposition d'un schéma de distribution des simulations basé sur une structure de graphe, et l'introduction de *Modes de synchronisation* permettant de définir et d'analyser différentes méthodes d'interactions entre agents exécutés sur différents processus. Ces travaux ont également donné lieu à l'implémentation d'un prototype de plateforme de simulation distribuée basée sur ces concepts, FPMAS.

Nous cherchons à étendre ces résultats avec l'étude d'une architecture de simulation distribuée de SMA générique, une analyse formelle des *Modes de synchronisation* déjà établis, et l'amélioration du prototype vers un outil plus robuste.

L'objectif de cette première partie consiste à définir le cadre de notre étude ainsi que les problématiques et questions de recherche associées dans le chapitre 1. Nous poursuivons au chapitre 2 par un état de l'art des méthodes relatives à la simulation de SMA et à leur exécution distribuée. L'analyse proposée des techniques et plateformes de simulations existantes tant dans le domaine du Calcul Haute Performance que des SMA permet de justifier l'intérêt de l'architecture logicielle générique proposée dans la seconde partie.

Chapitre 1.

Contexte et problématique

Nous introduisons dans ce chapitre les notions de simulation de SMA d'une part et de simulation distribuée d'autre part. Nous précisons ensuite nos objectifs en termes de simulation distribuée de SMA, issue de l'union de ces deux domaines. Enfin, nous définissons les problématiques associées ainsi que le plan du mémoire.

1.1. Simulation de Systèmes Multi-Agents

Il est possible de définir simplement un SMA par un ensemble d'entités autonomes en interaction avec leur environnement. Les SMA sont souvent utilisés pour décrire, simuler et analyser des systèmes dits *complexes*, dont le comportement global est par définition difficile à prédire en raison des interactions entre les nombreux composants du système.

Un SMA peut être biologique, physique ou informatique, réel ou virtuel. Dans ce contexte, les usagers d'un centre commercial, les êtres vivants d'une forêt, les véhicules et les piétons dans une ville, les cellules d'un être vivant ou les machines sur un réseau peuvent faire l'objet d'une description Multi-Agents. Certains systèmes peuvent également être définis de manière théorique, par exemple pour évaluer diverses stratégies dans le cadre de la microéconomie et de la théorie des jeux.

La modélisation Multi-Agents consiste en la définition plus ou moins formelle des agents, de l'environnement et des règles qui régissent les interactions entre les composants d'un système. La complexité des modèles est très variable, néanmoins l'intérêt de la modélisation Multi-Agents réside souvent dans le fait d'utiliser des règles simples afin de générer des phénomènes émergents complexes. La description minimaliste de règles d'alignement, de répulsion et d'attirance par rapport aux voisins dans une nuée d'oiseaux ou un banc de poissons suffit par exemple à expliquer et reproduire des mouvements globaux et complexes observés dans la nature.

Les travaux présentés ici traitent de la simulation numérique des modèles Multi-Agents, afin notamment d'en prédire l'évolution dans le temps.

1.2. Simulation distribuée

Compte tenu des exemples déjà cités, il est clair que la définition d'un SMA peut impliquer l'existence de millions d'agents ou plus, sur des échelles de temps très extensibles, que ce soit pour simuler l'économie européenne sur un trimestre ou l'activité

microbienne d'un champ pendant une heure. La simulation de tels systèmes devient rapidement impossible sur des ressources de calcul classiques. En effet, l'exécution séquentielle d'une simulation est limitée en termes de mémoire et peut produire des temps d'exécution irréalistes. La parallélisation en mémoire partagée, qui consiste en l'exécution d'une simulation avec plusieurs cœurs de processeurs ayant accès à une mémoire commune permet de résoudre en partie le problème du temps d'exécution. Le nombre de cœurs CPU (« Core Processing Unit ») pouvant accéder à une mémoire commune est actuellement limité d'un point de vue matériel : si les architectures à 4, 8 ou 16 cœurs se sont aujourd'hui démocratisées, des nombres plus élevés sont réservés à des machines spécialisées. Il est alors possible d'avoir recours au Calcul Haute Performance (HPC, « High Performance Computing ») grâce à des architectures dites *distribuées*, communément appelées *super-calculateurs* ou *clusters de calcul*. Cette technique consiste à multiplier à la fois la mémoire et les processeurs disponibles en exécutant les simulations sur des machines indépendantes connectées en réseau, ce qui accroit considérablement le potentiel de ressources disponibles : de nombreux super-calculateurs affichent ainsi des nombres de cœurs de processeurs de l'ordre de la dizaine de milliers. La simulation de SMA semble particulièrement adaptée à l'utilisation d'un tel nombre de cœurs grâce à la taille des modèles et au parallélisme intrinsèque des agents. La parallélisation d'une simulation en mémoire partagée constitue cependant un défi de taille car elle nécessite par exemple la gestion efficace des écritures concurrentes et des schémas d'exécution qui diffèrent du cas séquentiel. L'exécution en mémoire distribuée ajoutent davantage de problèmes pratiques et théoriques qui limitent l'application du Calcul Haute Performance à la simulation de SMA.

La conception et la programmation d'algorithmes non distribués sont généralement basées sur l'utilisation de variables accessibles de manière triviale dans la mémoire partagée. En revanche l'accès potentiellement concurrent à une variable localisée sur une machine distante, accessible seulement via un réseau, pose le problème de la *synchronisation des données* entre les machines. De telles contraintes nécessitent une conception radicalement différente des programmes et algorithmes, ainsi qu'un niveau d'expertise peu répandu parmi les concepteurs de modèles Multi-Agents. En effet, même si l'autonomie des agents rend leur exécution naturellement parallèle, les SMA se caractérisent par un grand nombre d'interactions parfois complexes entre les agents du système. Ainsi les interactions entre agents exécutés sur différentes machines peuvent se trouver sévèrement limitées, erronées, et dans tous les cas difficiles à mettre en place par les développeurs de simulations de SMA. D'où l'intérêt de concevoir des interfaces logicielles accessibles pour que la communauté puisse facilement bénéficier de l'utilisation de ressources de Calcul Haute Performance, en limitant au maximum les compétences nécessaires par rapport aux outils utilisés sur les architectures classiques.

De plus, les gains en performance liés à la distribution d'une simulation ne sont pas automatiques. En effet, le surcoût en termes de temps et de mémoire nécessaires pour pallier la distribution peut s'avérer tel que l'exécution distribuée est moins efficace qu'en mémoire partagée, même si le nombre de cœurs utilisés est plus important. Il est donc nécessaire de mettre en place des stratégies de distribution adaptées, de s'assurer de l'efficacité des outils mis à disposition, et d'identifier clairement leurs limites.

L'exécution distribuée d'une simulation soulève enfin des problèmes liés aux résultats des simulations. La reproductibilité des résultats n'est par exemple pas toujours garantie en fonction du nombre de processeurs utilisés. En effet, l'ordre global d'exécution des agents peut avoir un impact sur les résultats d'une simulation. Or, si cet ordre est déterministe dans le cas séquentiel, il ne l'est pas dans le cas du parallélisme car l'ordre d'exécution de deux agents exécutés sur des processeurs distincts dépend de la vitesse d'avancement de chaque processeur, qui peut varier selon des facteurs externes, matériels et stochastiques. Les méthodes de synchronisation entre les processus peuvent aussi impacter la qualité voire la validité des résultats. Pour toutes ces raisons, un critère de validité basé sur la correspondance avec une simulation séquentielle peut perdre son sens dans un contexte distribué.

1.3. Objectifs de la thèse

L'objectif général de nos travaux consiste à identifier, analyser et résoudre les différents problèmes génériques auxquels toute simulation distribuée de SMA doit faire face, à la fois grâce à une étude de la littérature et des plateformes existantes et grâce à l'expérience de développement d'une plateforme de simulation distribuée de SMA adaptée aux objectifs suivants : l'ergonomie, l'efficacité et la validité des résultats.

Ergonomie

L'ergonomie est un critère essentiel de développement visant à faciliter l'utilisation et l'extensibilité des plateformes logicielles. Cet aspect est particulièrement important dans notre étude qui vise à faciliter la distribution de simulations de SMA pour des non-experts en Calcul Haute Performance. Les solutions proposées doivent ainsi minimiser la complexité d'implémentation des modèles par l'utilisateur final, notamment grâce à un haut niveau d'abstraction des problèmes liés à la distribution. De plus, les solutions proposées doivent permettre l'exécution distribuée de la plus large gamme de simulations de SMA possible, sans qu'il ne soit nécessaire de modifier les règles d'un modèle ou d'implémenter de nouvelles fonctionnalités complexes en dehors des spécificités du système simulé.

Efficacité

L'objectif de toute distribution de simulation consiste à diminuer les temps de calcul ou à augmenter la quantité de mémoire disponible par rapport à une exécution en mémoire partagée, ou tout du moins par rapport à une exécution séquentielle. Il est donc nécessaire de mettre en place des solutions qui bénéficient réellement des cœurs de processeurs et des espaces mémoires mis à disposition par chaque machine du système distribué.

Validité des résultats

Les résultats fournis par une simulation doivent être conformes aux règles qui décrivent le SMA. Mais d'autres critères, qui dépendent généralement du contexte du modèle et des

besoins de l'utilisateur, comme la reproductibilité entre deux exécutions d'une simulation, sont parfois souhaitables et vont impliquer différents niveaux de contraintes.

1.4. Problématique

Nos contributions consistent alors à apporter des éléments de réponse aux questions de recherche suivantes :

1. Quelles sont les difficultés à surmonter pour permettre la simulation distribuée de SMA ? Fait-on face aux mêmes difficultés pour tous les types de modèles ?
2. Quel est le niveau de satisfaction des objectifs d'ergonomie, d'efficacité et de validité des résultats atteints par les solutions proposées dans les plateformes de simulation distribuées existantes ? Quelles sont les limitations selon les types de modèles ?
3. Comment mettre en place des solutions adaptables à tout type de modèle pour atteindre ces objectifs ?

Nous accordons dans nos contributions une importance particulière à l'étude pratique et théorique du problème de la synchronisation des données, dont la résolution constitue un point critique dans la simulation distribuée des SMA.

1.5. Plan du mémoire

Nous poursuivons cette partie introductive avec le chapitre 2, dans lequel nous proposons notamment une revue des techniques de simulation distribuée de SMA, au regard des objectifs précédemment définis. Ce chapitre est également l'occasion d'identifier, au fil des sections, les problèmes génériques auxquels toute simulation distribuée de SMA doit faire face. Dans la partie II, nous proposons une architecture générique facilitant la résolution de ces problèmes pour mettre en place des simulations distribuées de SMA dans un contexte matériel et logiciel arbitraire. Ces méthodes génériques sont l'aboutissement de notre expérience de développement de la plateforme FPMAS, dans laquelle les solutions proposées ont été implémentées. Le chapitre 3 se focalise en particulier sur le schéma de distribution d'une simulation de SMA et les problèmes génériques associés à la distribution comme le maintien de la continuité des données. Dans le chapitre 4, nous nous intéressons à la conception d'une interface générique d'équilibrage de simulation distribuée de SMA, ainsi qu'à l'implémentation de cette interface avec plusieurs méthodes. Une étude expérimentale basée sur la définition d'un *Méta-Modèle* permet d'évaluer leurs performances selon différents types de modèles Multi-Agents. Nous proposons dans le chapitre 5 une analyse du problème de la synchronisation des données, ainsi qu'une étude qualitative et quantitative des solutions proposées grâce au *Méta-Modèle* et à un modèle épidémiologique, le modèle *Virus*. Nous concluons enfin par un bilan permettant d'apporter des réponses à nos questions de recherche.

Chapitre 2.

État de l'art

Ce chapitre vise à dresser un état de l'art de la simulation distribuée de SMA. La définition des modèles considérés dans cette étude est présentée dans la section 2.1, avant de poursuivre avec la section 2.2 sur des exemples jugés pertinents dans le contexte de la simulation séquentielle. Nous abordons la simulation distribuée dans la section 2.3 avec une description des différents types d'architectures de calcul distribuées.

Les sections 2.4 à 2.7 sont focalisées sur des solutions à des problèmes liés à la simulation distribuée de manière générale, en prenant soin de faire le lien avec le cas des SMA. L'identification de ces problèmes, auxquels toute simulation distribuée de SMA doit faire face, est en soi une contribution, également issue de notre expérience de développement de la plateforme FPMAS.

Nous présentons pour finir une revue non exhaustive de plateformes de simulation distribuée de SMA dans la section 2.8, afin de les confronter aux objectifs définis au chapitre 1 et d'analyser leurs solutions aux problèmes identifiés dans ce chapitre.

Table des matières

2.1. Systèmes Multi-Agents

Afin d'établir le contexte de notre étude, nous commençons par introduire les éléments clés permettant de définir les SMA réels ou virtuels et de les distinguer d'autres techniques de modélisation.

2.1.1. La notion d'agent

La définition d'un *agent* est un problème de recherche en soi. La définition fournie dans la célèbre référence *Artificial Intelligence : A Modern Approach* (AIMA) [115] peut constituer un point de départ :

> Un agent est tout ce qui peut être vu comme percevant son environnement au travers de capteurs et agissant sur cet environnement au travers d'actionneurs. [1]

Cette définition évoque les *perceptions* et les *actions* effectuées par un agent, mais n'aborde pas explicitement la notion d'intelligence de l'agent, ce concept quasi philosophique pouvant faire l'objet de débats [57].

Cependant, la généricité de cette définition est appréciable : en effet, elle s'applique aussi bien aux agents les plus triviaux, qui réagissent directement à l'environnement sans délibération (agents *réactifs*, voire automates), qu'aux agents beaucoup plus complexes capables de raisonner et d'apprendre (agents *cognitifs*).

Une définition plus précise est proposée par Macal [81] grâce à quatre concepts, par ordre de complexité :

1. Individualité : les agents agissent indépendamment selon un comportement prédéfini.
2. Autonomie : les agents sont capables d'agir en fonction de leur environnement.
3. Interactivité : les agents sont capables d'échanger des informations avec les autres agents.
4. Adaptabilité : les agents sont capables d'adapter leurs comportements au cours de la simulation, par exemple grâce à des mécanismes d'apprentissage.

L'auteur insiste cependant sur la flexibilité et la non exhaustivité de cette définition. Le niveau de complexité des agents est également variable : si l'individualité de l'agent semble essentielle à sa définition, tous les agents ne sont pas adaptables.

Ces définitions sont applicables à la fois aux agents simulés, aux agents virtuels, ou à toute autre entité du monde physique en interaction avec son environnement, dont la nature varie selon le contexte.

Nous introduisons enfin la notion d'*état* des agents, constitué d'un ensemble de *propriétés* pouvant influencer le comportement de l'agent ou celui des autres. Ce concept peut se rapprocher des notions de *connaissance* ou de *croyance* des agents. Nous pouvons alors distinguer l'état réel d'un agent et l'état perçu par les autres, considéré comme une croyance. Cette subtilité n'est cependant pas nécessaire à la suite du raisonnement.

1. « An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. »

2.1.2. Systèmes complexes

L'autonomie des agents et les nombreuses interactions entre eux permettent l'émergence de propriétés et de comportements globaux qui ne peuvent être observés à l'échelle locale, d'où la caractérisation des SMA en tant que *systèmes complexes*. La modélisation Multi-Agents se caractérise ainsi par une approche « bottom-up », où on cherche à observer les propriétés émergentes à l'échelle globale d'un système en commençant par modéliser des comportements locaux et individuels [81]. La diversité des agents, leur capacité de prise de décision et la complexité de leurs interactions, souvent caractérisées par des comportements conditionnels à l'échelle des individus, rendent difficiles voire impossibles la description et la prédiction de l'évolution de ces systèmes par des systèmes d'équations, comme cela peut être le cas pour simuler certains phénomènes physiques (propagation de chaleur, écoulement d'un fluide, mouvements de particules...). Ainsi les domaines d'application de la modélisation Multi-Agents incluent la biologie [62, 86, 85, 87, 22, 66, 130], la sociologie [40, 117, 53, 19, 64], la gestion des réseaux d'énergie à l'échelle locale [17, 104, 102, 71, 20, 16], l'épidémiologie [61, 36, 75, 21], la simulation urbaine [82, 65] ou encore l'économie [52, 49, 39].

À noter qu'il existe des méthodes qui consistent à coupler des modèles microscopiques avec des systèmes d'équations [62, 85, 21] afin de limiter la granularité de la simulation tout en observant des phénomènes émergeant des comportements individuels. L'objectif consiste donc toujours à décomposer le modèle jusqu'à atteindre un niveau assez fin pour pouvoir décrire les entités et les interactions grâce à des règles individuelles simples à implémenter [46].

2.1.3. Environnement

L'environnement est un concept courant dans la définition des SMA. Dans leur livre *Growing Artificial Societies*[53] consacré à l'élaboration d'un SMA représentant une « société artificielle », Epstein et Axtell définissent l'environnement comme suit :

> [...] "l'environnement" est un milieu séparé des agents, *sur* lequel les agents opèrent et *avec* lequel ils interagissent. [2]

Leur exemple utilise en particulier une grille représentant un paysage dans lequel se trouvent des ressources exploitables par les agents, mais ils précisent que l'environnement pourrait être une entité plus abstraite, comme un réseau de communication dynamique.

L'environnement induit généralement des propriétés aux agents, telles que leur position, leur orientation, des liens de communications ou les plus proches voisins. L'utilisation d'une grille discrète pour représenter l'environnement est commune à une très large catégorie de modèles Multi-Agents [130, 53, 64, 117, 129], notamment en raison de sa simplicité d'implémentation et de l'utilisation de règles spécifiques à cette représentation, comme la capacité maximale de chaque cellule (ou toute autre propriété) ainsi que l'interaction avec les agents dans le voisinage de Moore ou de Von Neumann.

2. « [...] the "environment" is a medium separate from the agents, *on* which the agents operate and *with* which they interact. »

Il existe d'autres modèles où la position des agents est décrite de manière continue, comme le classique modèle des bancs de poissons [66]. Les modèles à base de GIS (« Geographic Information System ») consistent à définir la position des agents par des coordonnées géographiques.

Si les environnements à base de graphes peuvent être relativement abstraits (réseau de communication [50], réseau social [23, 103], ...), certains représentent un environnement géographique réel, comme un réseau urbain. Le modèle ChiSIM [82] représente un cas d'utilisation de graphe urbain, où les nœuds du graphe représentent les lieux de Chicago sur lesquels les agents peuvent se situer. Dans ce modèle, les agents se déplacent instantanément d'un nœud à l'autre. Il est également possible d'utiliser un graphe pour représenter précisément les routes, comme c'est le cas avec la base de données géographiques libre *OpenStreetMap*. Les agents peuvent alors se déplacer dans le graphe en étant associés à une position continue le long d'un arc ou dans un nœud sur lequel ils se trouvent. Combinée avec des modèles de suivi de véhicules [15], cette technique permet notamment de simuler avec précision le trafic routier à l'échelle d'une ville pour par exemple évaluer la congestion ou les émissions de pollution. Le simulateur SUMO [77] utilise notamment cette approche. Le simulateur MATSim [65] utilise une approche intermédiaire : les véhicules sont stockés dans une file au niveau de chaque nœud du réseau routier, et passent directement au nœud suivant dans leur trajectoire avec un certain délai en fonction de la longueur des arcs et de la congestion.

Les classes de modèles où les agents sont associés à une position explicite, qu'elle soit discrète, continue ou sur un graphe, sont qualifiées de "modèles spatiaux". La plateforme GAMA [123] est notamment spécialisée dans la simulation de ce type de modèles, qui implique des fonctionnalités communes telles que le déplacement des agents ou la notion de champ de perception.

2.2. Simulation de Systèmes Multi-Agents

Les éléments précédents permettent de définir de manière empirique un SMA. Il peut être nécessaire de *modéliser* le système de manière formelle, afin par exemple d'en effectuer la *simulation* pour en extraire des données ou observer des propriétés émergentes. La modélisation n'est pas directement et nécessairement liée à une technique de simulation particulière. Il existe plusieurs moyens de modéliser un SMA donné, et potentiellement plusieurs techniques pour simuler chaque modèle. Dans la pratique, les méthodes de modélisation sont régulièrement développées en conjonction avec une *plateforme de simulation* dédiée, chargée de l'animation des modèles. Nous présentons dans la suite une revue non exhaustive de techniques de modélisation et de plateformes de simulation non distribuées qui se sont révélées pertinentes dans le cadre de notre étude, de par leur réputation ou l'adaptabilité de certains concepts à la simulation distribuée. Nous nous focalisons notamment sur trois aspects : l'avancée temporelle de la simulation, la gestion de l'environnement et la spécification du comportement des agents.

Il existe deux catégories de techniques pour gérer l'avancée temporelle d'une simulation : la simulation par pas de temps, et la simulation à évènements discrets.

La première, popularisée par exemple par NetLogo, consiste à exécuter les comportements des agents uniquement à des dates prédéfinies. Il est possible, au cours d'un pas de temps, de planifier l'exécution de comportements aux pas de temps suivants, par exemple en ajoutant des agents à la simulation. La suppression des agents est possible, ce qui engendre l'annulation de l'exécution des comportements associés. Ce schéma d'exécution peut se généraliser en autorisant la planification dynamique de comportements à des dates arbitraires, comme c'est le cas avec Repast ou MASON.

La seconde se base sur une liste d'évènements datés à traiter dans l'ordre afin de faire évoluer l'état du système, sans qu'aucun changement n'ait lieu entre deux évènements consécutifs. De nouveaux évènements peuvent éventuellement être planifiés dans le futur lors du traitement d'un évènement. La notion de temps peut alors être perçue comme une grandeur relative grâce à l'ordre des évènements [74], un évènement a étant considéré comme précédent b si b peut dépendre de a. La simulation par pas de temps peut ainsi être conçue comme un cas particulier de simulation à évènements discrets, tous les évènements d'un pas de temps dépendant directement de ceux ayant eu lieu aux pas de temps précédents. Le langage de programmation orienté agent SARL [110] est un exemple de formalisme qui représente les interactions entre agents comme des évènements : les agents réagissent à des évènements et en émettent d'autres, indépendamment de la notion de pas de temps.

La majorité des plateformes recensées dans la littérature sont basées sur la simulation par pas de temps, généralement plus intuitive en termes de modélisation et de simulation. C'est notamment le cas pour NetLogo [10], Repast [96], MASON [79], MaDKit [63] et GAMA [123].

Certaines plateformes se distinguent par leur gestion de l'environnement. NetLogo permet de simuler des modèles à base de grille discrète, mais il est également possible de relier les agents entre eux pour former un graphe. Repast et MASON permettent de définir plusieurs types d'environnements (respectivement « projections » et « fields ») discrets, continus ou à base de graphe. Un agent peut appartenir simultanément à plusieurs environnements. La plateforme GAMA est quant à elle focalisée sur la simulation de modèles spatiaux et à base de GIS. À noter que dans NetLogo, l'environnement fait l'objet d'une représentation distincte des agents : les « patchs ». Dans MASON, Repast et GAMA, l'environnement n'est qu'un moyen d'associer une position aux objets, et les éléments qui s'y trouvent sont représentés par un type d'agent à part entière, éventuellement passif.

Outre ces contraintes environnementales, la modélisation du comportement des agents est relativement libre pour ces plateformes. Des techniques de modélisation plus spécifiques existent néanmoins. La modélisation IODA [73] ainsi que l'outil de simulation associé JEDI se focalisent par exemple sur la description des interactions entre agents pour construire des modèles Multi-Agents. MaDKit se base quant à lui sur la modélisation « Agent/Group/Role »(AGR) [55]. La méthode « Influence Reaction Model »(IRM) [56] propose de limiter les actions des agents à l'émission d'*influence* sur les autres agents et l'environnement, qui *réagissent* ensuite aux influences exercées sur eux. La modélisation IRM a par la suite été étendue avec les modélisations « Inflence Reaction Model for Simulation »(IRM4S) [90] et « Influence Reaction Model for Multi-

Level Simulation »(IRM4SMLS) [94], ce dernier étant accompagné de la plateforme de simulation associée SIMILAR [93]. La méthode IRM permet notamment de résoudre efficacement les problèmes liés aux *actions simultanées*, par exemple dans le cas où deux agents cherchent à collecter le même objet au cours du même pas de temps.

Cette liste non exhaustive fait déjà état de diverses techniques de modélisation et de simulation. Pour des raisons de simplicité de conception et d'implémentation, nous restreignons nos travaux à la *simulation par pas de temps*. La grande majorité des modèles étudiés étant spécifiés par pas de temps, ce choix est peu restrictif. Aucune contrainte n'est imposée a priori à l'environnement ou à la spécification des comportements et interactions. Ainsi l'objectif est d'établir des méthodes de simulation distribuée au niveau le plus générique possible, sans chercher à définir de nouvelles méthodes de modélisation, afin de permettre l'exécution distribuée de tout type de modèles Multi-Agents.

2.3. Architecture de calcul distribuée

Afin d'introduire et de justifier le contexte de notre étude en termes de calcul distribué, nous présentons brièvement diverses architectures permettant l'exécution parallèle ou distribuée de simulations.

En effet, les plateformes précédemment citées sont généralement conçues pour une exécution séquentielle, ou tout du moins pour une exécution sur une seule machine. Il est alors possible de mettre en place une exécution parallèle multi-tâche (« multi-threading ») par exemple grâce à la librairie OpenMP [98]. L'exécution parallèle est alors dite en *mémoire partagée*, car toutes les tâches ont physiquement accès à un espace mémoire commun.

La simulation GPU offre également des opportunités de simulation large échelle et de parallélisation en mémoire partagée grâce à l'utilisation de plusieurs centaines voire milliers de cœurs, y compris dans le cadre de l'application à la simulation de SMA [76]. Ces architectures sont cependant difficilement généralisables en raison de la spécificité des instructions exécutées par ce type de processeurs, dont l'utilisation reste limitée en raison de l'irrégularité des comportements et interactions entre agents.

Les infrastructures de calcul *distribuées* donnent quant à elles accès à des ensembles de machines connectées entre elles sur un réseau. Cette méthode permet d'accroitre très largement le nombre de cœurs de processeur et la quantité de mémoire disponibles pour exécuter une simulation, la taille du système n'étant limitée que par le nombre de machines qu'il est possible de connecter sur le réseau, au prix d'une forte contrainte sur la conception des programmes. En effet, chaque machine du système (*nœud de calcul*) n'a pas d'accès direct et physique à la mémoire des autres. C'est-à-dire qu'une instance de simulation exécutée sur un nœud ne peut pas accéder directement à une variable stockée dans la mémoire d'une autre machine. Le seul moyen d'échanger des données est l'échange de messages explicites, par exemple grâce à la librairie MPI [95], ce qui modifie de manière drastique l'implémentation d'algorithmes initialement conçus pour une exécution en mémoire partagée. Dans le domaine du Calcul Haute Performance, de nombreux efforts sont menés pour mettre en place des structures de données distribuées [31], la

gestion distribuée d'adresses mémoires [132] ou même la définition de langages dédiés à l'exécution distribuée [54] avec pour objectif récurrent l'abstraction des difficultés spécifiques à la distribution. Bien qu'utile, l'exploitation de ces concepts ne suffit cependant pas à mettre en place la simulation distribuée des SMA ou de s'adapter à leurs spécificités.

Parmi les ressources de calcul distribuées, on distingue les clusters *hétérogènes* et *homogènes*. Dans un cluster hétérogène, il n'est pas garanti que toutes les machines possèdent les mêmes caractéristiques en mémoire et en puissance de calcul. C'est par exemple le cas avec les clusters de postes de travail (« workstations ») constitués de machines à usage générique connectées via un réseau classique, ou avec l'informatique en nuage (« Cloud Computing »), qui permet une allocation de ressources de calcul à la demande, en fonction des simulations à exécuter [47] (« Simulation as a Service »). Il est alors nécessaire de prendre en compte les différences entre les machines pour mettre en place une distribution efficace, ce qui peut mener à des solutions où chaque machine est associée à un rôle spécifique dans l'exécution de la simulation [126]. Les processus n'ont alors pas tous un rôle symétrique, ou peuvent faire preuve d'une forme de centralisation dans l'exécution ou la gestion des données (relations clients / serveurs ou maitres / esclaves). En comparaison, dans un cluster homogène, chaque machine joue le même rôle et possède les mêmes caractéristiques physiques. Cette architecture se retrouve notamment dans les clusters de calcul orientés Calcul Haute Performance, où les nœuds utilisés au cours d'une simulation ont les mêmes caractéristiques matérielles et sont reliés entre eux via un réseau à haut débit spécialisé. L'absence de spécificité de chaque nœud permet ainsi d'abstraire leurs caractéristiques physiques d'un point de vue algorithmique. Le fait que chaque machine joue le même rôle permet par ailleurs une plus grande extensibilité, les algorithmes étant conçus indépendamment du nombre de cœurs utilisés, sans considération sur les différences de performances ou les propriétés spécifiques de chaque machine.

Dans le cadre des clusters hétérogènes, on considère parfois les problèmes liés à la tolérance aux fautes, c'est-à-dire au maintien du service de simulation dans le cas de la panne d'une machine du cluster. Cette problématique n'est cependant pas pertinente dans le cas des clusters homogènes de Calcul Haute Performance, pour lesquels les durées des simulations sont négligeables par rapport à leur temps de fonctionnement sans panne.

Parmi ces possibilités, nous ne considérons dans nos travaux que l'exécution **distribuée** de simulation de SMA sur des clusters de calcul **homogènes** équipés de **CPU** : nous nous focalisons donc sur l'aspect générique et extensible des simulations distribuées de SMA, sans considération pour les problèmes de tolérance aux fautes. L'application de certaines solutions dans d'autres contextes et inversement reste possible, mais ne sera pas abordée explicitement. Nous étudions dans la suite l'exécution de simulation de SMA sur ce type d'architecture.

2.4. Distribution d'une simulation de SMA

Un *processus* est défini comme l'exécution d'une instance d'un programme informatique. Une simulation est dite séquentielle lorsqu'elle est assurée par un unique processus. La simulation distribuée se caractérise par l'utilisation de plusieurs processus parallèles, chaque processus n'ayant pas d'accès direct à la mémoire des autres. Ainsi il est d'abord nécessaire de déterminer le rôle de chaque processus dans l'exécution de la simulation globale. Le travail d'un processus dépend ensuite généralement de celui des autres, impliquant un accès partiel à leurs données : d'où la notion de *continuité des données*. Nous présentons ici une revue des techniques permettant de résoudre ces problèmes fondamentaux dans le cadre de la simulation distribuée de SMA.

2.4.1. Exécution

Dans tous les exemples qui ont été recensés dans la littérature [42, 44, 114, 105, 125, 24, 23, 82, 121, 25, ...] l'exécution distribuée d'une simulation Multi-Agents consiste à associer à chaque processus disponible un ensemble d'agents à exécuter, dans l'objectif de répartir la charge de calcul totale représentée par la simulation globale. Cette méthode ne doit pas être confondue avec l'exécution par lots (« batch »), qui distribue un ensemble de simulations, par exemple pour effectuer de l'exploration de paramètres. Dans ce cas, chaque processus exécute une simulation indépendante, et aucune interaction n'est nécessaire entre les processus pendant l'exécution de chaque simulation. L'objectif consiste ici à exécuter une seule simulation sur un ensemble de processus.

Quelle que soit la distribution, nous définissons à l'échelle d'un processus les agents *locaux* comme ceux exécutés par ce processus, et les agents *distants* comme ceux exécutés par d'autres processus. Chaque agent est ainsi *local* du point de vue d'un unique processus, et *distant* du point de vue de tous les autres. La distribution d'une simulation consiste alors simplement à assigner l'exécution d'un ensemble d'agents *locaux* à chaque processus. Il est ensuite nécessaire d'assurer la continuité des données pour permettre les interactions entre agents exécutés sur différents processus.

2.4.2. Continuité des données

Les interactions entre agents sont essentielles à tout modèle Multi-Agents. De plus, elles ont généralement lieu de manière stochastique et dynamique au cours de l'évolution du système. Indépendamment de la technique de distribution et de simulation mise en place, les agents *locaux* seront donc nécessairement amenés à interagir avec des agents exécutés par d'autres processus. Ainsi, afin d'assurer la continuité des données de la simulation, il est nécessaire d'ajouter aux agents *locaux* un ensemble de représentations d'agents *distants*, ou agents *délégués*, pour permettre la mise en place d'interactions entre les agents exécutés sur un processus et les autres. Un agent *délégué* permet au minimum l'accès à l'identifiant de l'agent *distant* qu'il représente, ainsi qu'à une référence au processus sur lequel il est actuellement exécuté.

La notion d'agent *délégué* se rapproche fortement des *espaces mémoire globaux*

partitionnés(« Partitionned Global Address Spaces », ou PGAS) [18, 132]. En effet, les PGAS permettent d'implémenter un système de pointeurs vers des adresses mémoire qui référencent implicitement des zones associées à des processus distants. En ce sens, il est possible de voir les agents *délégués* comme des pointeurs distants vers des agents *distants*, en assimilant les adresses des pointeurs aux identifiants des agents. Certaines librairies, comme BCL [31] ou de nombreux langages distribués tels que UPC [43], Coarray Fortran [97] ou X10 [37] se basent en partie sur les PGAS. Les PGAS en eux-mêmes ne tirent cependant pas partie des spécificités des SMA, en supposant notamment que n'importe quel processus peut être amené à accéder à n'importe quelle adresse, alors que les besoins d'accès des agents entre eux sont généralement limités par la notion de perception. De plus, les PGAS en eux-mêmes ne définissent pas les mécanismes d'accès aux données distantes [18], mais seulement l'accès à leur localisation.

Même si tous ne font pas référence explicitement au concept d'agent *délégué*, la totalité des simulations distribuées de SMA recensées dans la littérature utilise au moins implicitement ce concept. Se pose alors la question de la construction de l'ensemble d'agents *délégués* à représenter sur le processus.

Certaines approches permettent l'accès de chaque agent *local* à l'intégralité de l'environnement. Rao et Chernyakhovsky [106] présentent un exemple de simulation distribuée où une copie locale et complète de l'environnement est représentée sur tous les processus. L'accès à l'environnement s'effectue alors localement, et une mise à jour globale des copies est effectuée par des communications collectives. Or l'accès d'un agent *local* à son environnement fait l'objet de contraintes similaires à l'accès aux agents *distants*. Ainsi il est possible d'avoir, sur chaque processus, un ensemble d'agents *délégués* représentant la totalité des agents *distants*. Cette méthode possède l'avantage de grandement simplifier la création et la suppression d'agents *délégués* au cours de l'évolution de la simulation, mais pose cependant rapidement des problèmes d'extensibilité pour des raisons de consommation de mémoire. Dans la même idée, certains travaux [84, 83] proposent de gérer l'accès aux cellules de l'environnement via un fichier partagé par tous les processus de la simulation. Les opérations de lecture et d'écriture dans un fichier, plus coûteuses que des accès directs dans la mémoire vive (RAM), peuvent cependant poser des problèmes de passage à l'échelle compte tenu du grand nombre de lectures et d'écritures concurrentes réalisées par les agents.

Il existe des solutions plus conventionnelles proposées par les plateformes de simulations existantes, explicitées dans la section 2.8 et détaillées formellement dans la partie II, qui tendent à réduire la quantité d'agents *délégués*.

Dans la suite, afin d'assurer la continuité des données et l'exécution cohérente de la simulation, nous supposons que l'ensemble des agents *délégués* assignés à chaque processus permet aux agents *locaux* d'interagir avec les agents exécutés sur d'autres processus en accord avec les règles définies par le modèle simulé. Nous ne considérons donc pas la possibilité de tronquer le champ de perception des agents à la frontière des processus par exemple.

Toute exécution distribuée de simulation de SMA ainsi définie doit encore faire face à plusieurs problèmes, dont les solutions recensées dans la littérature sont présentées aux sections suivantes. On constate cependant une absence de formalisme permettant

de clairement identifier les problèmes liés à la distribution, ce qui limite la validation et la comparaison des solutions proposées. Nous proposons dans la suite une description introductive des problèmes de synchronisation des données, de synchronisation temporelle et d'équilibrage de charge, afin notamment de permettre l'analyse des plateformes de simulation distribuée de SMA existantes.

2.5. Synchronisation des données

La continuité des données telle qu'elle a été définie précédemment permet aux agents *locaux* d'avoir un accès aux agents *distants* grâce aux agents *délégués*, nécessaires pour mettre en place les interactions requises. La possibilité d'accès à un agent *distant* ne suffit cependant pas à définir les modalités d'accès à cet agent, ou les types d'interactions autorisés. De manière plus générale, le problème de la *synchronisation des données* correspond aux modalités d'accès d'un processus aux données des autres de manière cohérente. La nature de ces accès varie grandement selon les solutions proposées, introduisant diverses contraintes de modélisation par rapport aux exécutions séquentielles ou en mémoire partagée. On observe également des différences significatives dans les besoins en interactions des modèles, les rendant plus ou moins compatibles avec les contraintes de synchronisation imposées par les plateformes. Il est notamment possible d'imaginer différents niveaux de contraintes liées à la synchronisation des données. Nous nous limitons ici à une description des modalités d'interactions observées dans certains modèles. Le chapitre 5 fait l'objet d'une étude formelle et détaillée, notamment grâce à la définition de *modes de synchronisation*.

2.5.1. Lectures

Le niveau le plus simple consiste, lors de l'exécution du comportement d'un agent, en un accès en lecture seule aux données des autres. C'est par exemple le cas avec le modèle classique des nuées d'oiseaux (« flocking ») [11], où il suffit à chaque agent de lire la position de ses voisins pour se déplacer. Une version en lecture seule d'un modèle épidémiologique peut être implémentée en mettant à jour l'état d'infection des agents en fonction de l'état de leurs voisins. Les lectures peuvent se faire selon deux méthodes, qui varient selon la temporalité de l'accès aux données. La première consiste à accéder aux données du pas de temps en cours : l'état de l'agent voisin dépend alors de l'ordre d'exécution des agents. La seconde consiste à lire les données à partir d'une copie de l'état des agents au pas de temps précédent (ou copie *ghost*) : les données lues sont alors indépendantes de l'ordre d'exécution, ce qui permet d'atteindre un plus haut niveau de reproductibilité. À noter que de telles règles ne s'appliquent pas seulement à l'accès des données des agents *distants*, mais peuvent aussi s'appliquer aux interactions entre agents *locaux*, d'où l'utilisation possible d'un *ghost* en simulation séquentielle [88].

2.5.2. Écritures non concurrentes

On observe, dans certains contextes, l'utilisation d'écritures non concurrentes, où de multiples écritures n'engendrent ni conflits ni résultats incohérents même sans forcer l'accès exclusif à l'état d'un agent pour effectuer les écritures.

C'est notamment le cas pour les écritures que nous qualifions d'*idempotentes*. Les écritures de ce type sur l'état d'un agent par les autres sont possibles et doivent être prises en compte, mais il est garanti que les écritures suivantes ne changeront pas l'état de l'agent. C'est par exemple le cas pour un modèle épidémiologique où l'infection d'un agent voisin représente une écriture, mais où l'état final de l'agent infecté ne dépend pas du nombre d'agents l'ayant infecté.

De manière plus générale, une écriture peut être considérée comme non concurrente quand sa réalisation par un agent ne peut altérer le comportement des autres au cours du pas de temps, et ne génère donc pas de conflit. C'est par exemple le cas avec l'envoi d'un message à un agent. En effet, supposons que l'état d'un agent est défini par un conteneur auquel les autres agents peuvent ajouter des messages. L'ajout d'un message va modifier l'état de l'agent, ce qui représente bien une écriture. Dans un contexte de parallélisme en mémoire partagée, il peut certes être nécessaire de gérer la concurrence d'accès au conteneur, par exemple grâce à des opérations d'ajout atomiques, afin d'assurer la cohérence de la mémoire. Cependant, du point de vue du modèle Multi-Agents, en supposant que l'ordre des messages n'importe pas, il est possible de considérer l'opération comme non concurrente car les agents ne sont pas en conflit sur une ressource : l'ajout d'un message par un agent ne peut empêcher les autres de le faire. Quels que soient le nombre et l'ordre d'ajout des messages, à la fin du pas de temps, la même liste de messages est stockée dans le conteneur.

Comme nous allons le voir en partie II, de telles écritures possèdent notamment l'avantage de pouvoir être mises en attente pendant un pas de temps avant d'être réellement effectuées en fin de pas temps, sans conflit, après l'exécution de tous les agents, ce qui n'est pas possible avec les écritures concurrentes.

2.5.3. Écritures concurrentes

Dans le cas des écritures concurrentes, la réalisation d'une écriture par un agent doit nécessairement être prise en compte immédiatement par les autres, influençant leurs comportements, afin d'assurer le respect des règles du modèle et donc l'exécution cohérente de la simulation.

Considérons par exemple un modèle où les agents peuvent se voler une ressource entre eux, la ressource ne pouvant être partagée entre plusieurs agents. Soient trois agents, A_0, A_1 et A_2, où A_0 possède la ressource à voler. Au cours du pas de temps, A_0 ne fait rien, et A_1 et A_2 vont tenter de voler la ressource à A_0 : prendre la ressource à A_0 représente un changement d'état de A_0, et donc une écriture. Cependant, contrairement au cas des écritures non concurrentes, la réalisation d'une écriture doit être prise en compte par l'autre agent car un seul peut voler la ressource. Par exemple, si A_1 vole la ressource, A_2 doit adapter son comportement pour ne pas la prendre par la suite. Il est

donc nécessaire d'assurer un accès exclusif à l'agent qui vole la ressource pour garantir que personne n'a déjà pris la ressource, et que personne ne puisse la prendre suite au vol. Le modèle classique du Proie-Prédateur [130, 14] est aussi un exemple de modèle ayant besoin d'écritures concurrentes, deux prédateurs ne pouvant manger la même proie.

Le problème de gestion de la concurrence est implicitement et trivialement résolu dans le cas d'une exécution séquentielle, mais doit être résolu explicitement dans le cas où les agents sont exécutés en parallèle sur des nœuds de calcul distants. De plus, la mise en place d'algorithmes de « Readers-Writers » par exemple est relativement simple et classique en mémoire partagée, mais pose davantage de difficultés avec les processus distribués, l'accès aux données et leur verrouillage ne pouvant s'effectuer que par envoi de messages explicites.

Dans le cas où plusieurs processus ont besoin de l'accès en écriture simultané à plusieurs ressources, des blocages (« deadlocks ») peuvent survenir. Ces problèmes d'*allocations de ressources* sont difficiles à résoudre y compris en mémoire partagée.

Dans le cadre de la simulation de SMA, d'autres travaux se basent sur la gestion centralisée des conflits [116, 103], mais ces méthodes posent des problèmes de généricité, de passage à l'échelle, et rendent possible l'échec de la demande d'action d'un agent, ce qui peut avoir un impact sur la modélisation du système étudié. Une méthode de négociation et de résolution de conflits décentralisée [107] a été abordée dans le cadre de FLAME GPU, mais la solution proposée nécessite de nombreuses barrières de synchronisation entre les processus.

2.5.4. Nécessité des écritures

Il est légitime de se questionner sur la nécessité réelle des écritures dans la définition des modèles. En effet, est-il possible de reformuler un modèle ayant des besoins en écritures en un modèle effectuant seulement des lectures, afin d'en simplifier la distribution, sans modifier les règles et donc les résultats du modèle ?

Le cas des écritures non concurrentes y semble propice. En effet, plutôt que réaliser une écriture non concurrente, l'agent cible peut lire les données depuis les agents susceptibles d'effectuer les écritures. Pour l'exemple de l'envoi de message, les agents peuvent stocker localement les messages à envoyer, et le récepteur peut les lire et les ajouter à son conteneur. Le résultat à la fin de chaque pas de temps est alors le même qu'avec les écritures.

Plus généralement, tout modèle spécifié par « Influence-Reaction » [56] peut être implémenté en lecture seule : il suffit de stocker localement l'influence et les agents réagissent aux influences lues sur les agents voisins.

Le cas des écritures concurrentes peut parfois être résolu avec une modélisation de type « Influence-Reaction » : les agents émettent une influence correspondant au souhait d'acquérir une ressource et, en réaction, un mécanisme de résolution de conflit exécuté au niveau de l'agent qui possède la ressource permet de l'attribuer de manière cohérente. Cependant, reformuler un modèle de la sorte peut impliquer un changement significatif des règles du modèle. Par exemple, considérons le cas du modèle Proie-Prédateur. Dans la modélisation avec écritures concurrentes, un prédateur P mange une proie p si elle est

vivante. Dans le même pas de temps, un prédateur P' constate ensuite que la proie p n'est plus vivante, et il peut donc essayer d'attaquer une autre proie vivante dans son champ de perception. En revanche, dans une modélisation « Influence-Reaction », P et P' vont émettre le souhait de manger p. Supposons alors que le mécanisme de résolution de conflit attribue la capture de la proie à P. Le prédateur P' ne mange ensuite aucune proie alors que les règles du modèle d'origine lui permettaient d'essayer d'en attaquer une autre.

De telles discussions sur le meilleur choix de modélisation ne font pas l'objet de nos travaux. Cependant, la littérature, et notamment les nombreux exemples de la librairie NetLogo [12], contiennent des exemples de modèles dont la formalisation nécessite des écritures concurrentes. Dès lors, notre objectif est de proposer des méthodes permettant la simulation distribuée de ces modèles sans avoir à les altérer.

2.6. Synchronisation temporelle

La simulation distribuée de SMA peut faire l'objet de trois grands types de synchronisations temporelles [47] : *par pas de temps*, *conservative* et *optimiste*.

La synchronisation par pas de temps consiste simplement à exécuter tous les comportements des agents planifiés au pas de temps actuel, puis à passer au suivant. La planification des comportements pouvant être dynamique, le prochain pas de temps correspond à la date minimale à laquelle au moins un évènement est planifié. Le passage d'un pas de temps à l'autre est assuré par une barrière de synchronisation temporelle stricte[3]. En conséquence, si un seul comportement est planifié au pas de temps t_1 sur un processus p_0, tous les autres doivent attendre qu'il ait terminé avant de passer au prochain pas de temps $t_2 > t_1$. Cette technique peut s'avérer extrêmement coûteuse si peu d'évènements sont planifiés à chaque pas de temps.

Les synchronisations conservatives [92, 119] et optimistes [67, 48], qui supposent l'utilisation du paradigme de simulation par évènements discrets parallèle (« Parallel Discrete Event Simulation » ou « PDES ») [59], permettent de lever cette limitation. La simulation par pas de temps pouvant être conçue comme un cas particulier de simulation à évènements discrets, avec l'émission d'évènements d'un pas de temps à l'autre, il est théoriquement possible d'appliquer ces synchronisations à ce schéma d'exécution.

De manière générale, l'exécution distribuée d'une simulation à évènements discrets consiste à traiter les évènements en parallèle. Dans ce contexte, les synchronisations conservatives et optimistes consistent à relaxer les barrières de synchronisation strictes de la synchronisation par pas de temps pour seulement assurer la causalité à l'échelle globale de la simulation : en effet, il suffit que suite au traitement d'un évènement de date t aucun évènement de date $t' < t$ dont il dépend ne puisse être traité à posteriori pour assurer la cohérence de l'exécution de la simulation.

La synchronisation conservative consiste à déterminer, **à l'échelle de chaque processus**, la date minimale du prochain évènement à traiter afin d'avancer sans risque à cette date. Ainsi, lorsqu'un processus termine de traiter une liste d'évènements de date

3. Ce qui correspond par exemple au comportement de la méthode `MPI_Barrier` [95]

t_i, il ne peut commencer à traiter des évènements à une date $t_j > t_i$ que lorsqu'il est certain qu'aucun autre processus ne peut lui transmettre de nouveaux évènements à une date t_k telle que $t_i \leq t_k < t_j$. La valeur d'horloge de chaque processus est définie par la date du dernier évènement traité. Ainsi, contrairement à la synchronisation par pas de temps, il n'est pas garanti que tous les processus soient synchronisés sur la même horloge. En effet, un processus peut avancer jusqu'à la date t_j s'il est garanti qu'il ne recevra plus aucun évènement d'aucun autre processus avec une date $t_i < t_j$, même si les autres processus ne sont pas encore à la date t_j. Le processus peut donc éviter d'attendre que tous les autres aient terminé d'exécuter leurs évènements de dates inférieures à t_j si lui-même n'en a aucun à traiter. L'avancée globale de la simulation est assurée par le progrès du temps global virtuel (« Global Virtual Time », GVT) défini comme la valeur minimale de toutes les horloges locales. Dans le cas de la synchronisation par pas de temps, le GVT correspond trivialement au pas de temps actuel. En appliquant une synchronisation conservative à l'exemple par pas de temps précédent, s'il est garanti que le traitement par p_0 de l'évènement au pas de temps t_1 n'émettra pas d'évènement aux autres processus avec une date $t < t_2$, les autres processus peuvent alors passer au pas de temps t_2 sans attendre p_0.

La synchronisation par pas de temps est ainsi un cas particulier et non optimisé de synchronisation conservative. On observe que dans les deux cas, l'avancée globale de la simulation est limitée par le processus le plus lent.

La synchronisation optimiste tente de résoudre ce problème en autorisant chaque processus à traiter les évènements locaux sans attendre les autres processus. Le traitement de certains évènements peut cependant entrainer la violation du principe de causalité, lorsqu'un évènement de date t_k antérieur au dernier évènement traité est reçu (on parle de « message retardataire » ou « straggler message »). Il est alors possible de revenir en arrière (« rollback ») en annulant les effets des évènements traités avec une date postérieure à t_k pour revenir à un état cohérent. Le GVT correspond alors à la date à partir de laquelle tous les évènements ont été traités et donc au delà de laquelle aucun retour en arrière n'est possible. En appliquant une synchronisation optimiste à l'exemple précédent, tous les processus sont autorisés à avancer jusqu'à t_2 sans attendre p_0 ni aucun autre processus. Si le traitement de l'évènement à la date t_1 par p_0 produit par la suite des évènements avec une date $t_k < t_2$, tous les processus impactés par ces évènements doivent alors revenir en arrière jusqu'à la date t_k, en annulant le traitement des évènements avec une date $t > t_k$.

Fujimoto [58] propose une revue des algorithmes fondamentaux de synchronisation conservative et optimiste.

Les tentatives de mise en place de la synchronisation optimiste dans le cadre de la simulation distribuée de Systèmes Multi-Agents sont complexes, rares et datées [124, 100, 122, 105]. En effet, la synchronisation optimiste, en plus de sa complexité d'implémentation et des contraintes associées, comme la possibilité d'effectuer des retours en arrière ou le maintien d'un historique de l'état des agents, est peu viable dans le cadre de la simulation de SMA, où les dépendances de données entre les processus sont extrêmement importantes en raison du nombre d'agents et du nombre d'interactions entre eux, ce qui peut produire un grand nombre de retours en arrière à la chaine. De manière

plus générale, les synchronisations conservatives et optimistes ont été conçues pour les systèmes où peu d'évènements sont planifiés à chaque date [59], alors que les simulations de SMA impliquent généralement la planification d'un très grand nombre d'évènements à des dates fixes. La plupart des simulateurs distribués de SMA font donc le choix de la synchronisation par pas de temps.

D'autres travaux ont abordé la possibilité de s'affranchir de la causalité sur une fenêtre temporelle de taille limitée, en autorisant des interactions a priori incohérentes d'un point de vue temporel [109, 131]. En supposant que l'impact de ces interactions n'est pas significatif dans le cadre d'une exécution large échelle, il est possible de relaxer la synchronisation pour améliorer les performances. Une telle violation de la causalité semble cependant difficilement généralisable à toute simulation de SMA. L'utilisation de ces techniques doit donc au plus rester un choix pour le modélisateur.

2.7. Équilibrage de charge

L'exécution distribuée d'une simulation de SMA consiste à assigner à chaque processus l'exécution d'un ensemble d'agents. Il est donc nécessaire de mettre en place une méthode permettant de construire ces ensembles. Dans ce contexte, un algorithme d'*équilibrage de charge* (« load balancing ») est défini comme la fonction qui associe à chaque agent le processus sur lequel il sera exécuté au cours de la simulation. Nous appelons *partition* l'ensemble de sous-ensembles d'agents (*sous-partitions*) associés à chaque processus. Le *partitionnement* désigne le procédé de construction d'une *partition*. Dès lors, un algorithme d'équilibrage de charge est obtenu par l'application ponctuelle (équilibrage *statique*) ou itérative (équilibrage *dynamique*) du partitionnement.

Dans le cas statique, l'algorithme est appliqué en début de simulation et la même partition est utilisée pour toute la durée de la simulation. Dans le cas dynamique, l'algorithme peut être appliqué lorsqu'un seuil de déséquilibre est franchi ou à intervalles réguliers au cours de la simulation pour compenser les déséquilibres dus par exemple au déplacement des agents dans leur environnement, à la mort ou la naissance d'agents, ou à la mise à jour de contacts.

L'avancée d'une simulation distribuée basée sur la synchronisation par pas de temps est limitée par le processus le plus lent. Or le temps d'exécution d'un processus dépend du temps d'exécution total des agents qui lui sont associés. De plus, l'exécution distribuée implique l'existence d'un volume de communication entre chaque paire de processus. En effet, nous supposons que toute interaction entre deux agents assignés à différents processus va nécessiter un échange de messages. L'objectif de l'équilibrage de charge consiste donc à équilibrer les temps d'exécution des processus et à limiter les communications afin d'améliorer les performances de la simulation.

La mise en place de l'équilibrage de charge est cependant une question de compromis : le gain en performance induit par une partition de meilleure qualité doit compenser le temps de construction de cette partition. C'est pourquoi certains algorithmes se focalisent davantage sur la construction rapide d'une partition, quitte à perdre en qualité. La simplicité d'implémentation est parfois un autre critère à prendre en compte.

Pour résumer, les algorithmes d'équilibrage de charge peuvent chercher à atteindre tout ou partie des ces objectifs :

1. Équilibrage de la charge de calcul : l'algorithme cherche, pour chaque pas de temps, à minimiser les écarts de temps d'exécution entre les processus.
2. Minimisation des communications : toute communication étant coûteuse, on cherche à réduire au maximum les échanges entre les processus.
3. Temps de calcul : l'algorithme cherche à minimiser le temps d'exécution de l'algorithme d'équilibrage lui-même.
4. Facilité d'implémentation : les solutions les plus simples à mettre en place sont parfois privilégiées.

Afin d'estimer les temps d'exécution à équilibrer, chaque agent est associé à un poids représentant sa charge de calcul. Ce poids peut représenter un temps d'exécution réel, ou toute autre grandeur qui permet d'exprimer la charge de calcul d'un agent relativement à celle des autres. La charge totale associée à un processus est définie comme la somme des poids des agents qui lui sont associés.

Lui et Chan [78] se focalisent sur les objectifs de performance de la simulation en définissant formellement le problème de l'équilibrage de charge comme un problème d'optimisation cherchant à minimiser le coût d'une partition P définit de la façon suivante :

$$C_P = W_1 C_P^W + W_2 C_P^L \text{ avec } W_1 + W_2 = 1 \tag{2.1}$$

C_P^W représente le coût impliqué par le déséquilibre de charge entre les processus et C_P^L représente le coût associé aux communications. Les réels W_1 et W_2 permettent d'ajuster l'importance relative des deux critères à optimiser.

Par réduction au problème de la Somme des Sous-Ensembles [68], démontré comme étant NP-Complet [60], on peut montrer que le problème de décision associé à l'équilibrage de charge est lui-même NP-Complet. Dans l'hypothèse où $P \neq NP$, il n'existe donc pas d'algorithme avec une complexité temporelle polynomiale pouvant résoudre de manière exacte le problème de l'équilibrage de charge. Afin de minimiser le temps de calcul associé à l'algorithme d'équilibrage, les solutions proposées sont donc généralement approximatives.

Il existe de nombreuses méthodes d'équilibrage de charge, basées sur des concepts variés comme le découpage de l'environnement, le partitionnement de graphe ou la proximité spatiale des agents.

2.7.1. Méthodes par découpage de l'environnement

Dans le cas d'un environnement spatial continu ou discret, un algorithme simple à mettre en place consiste à associer une partie de l'environnement à chaque processus. Dans le cadre d'un environnement 2D, la décomposition peut se faire en bande ou en grille. La méthode est facilement extensible au cas 3D. Les agents sont systématiquement exécutés sur le processus associé à la partie de l'environnement où ils sont localisés, et peuvent être amenés à migrer entre les processus au cours de leurs déplacements

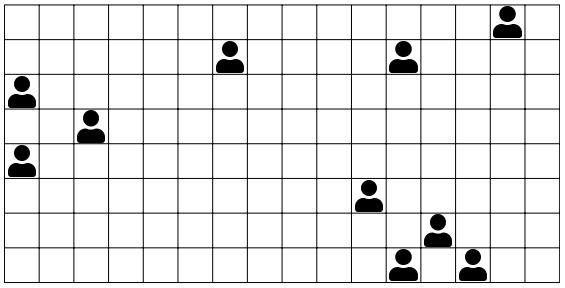


FIGURE 2.1. – Exemple d'environnement à base de grille.

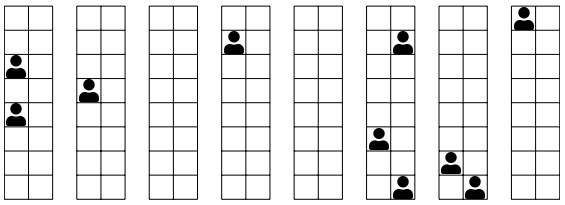


FIGURE 2.2. – Exemple de partition en bandes.

dans l'environnement. Un exemple d'environnement à base de grille est présenté sur la figure 2.1.

Un exemple de décomposition en bandes de ce modèle minimaliste sur 8 processus est présenté sur la figure 2.2. La décomposition en grille du même environnement avec le même nombre de processus est présenté sur la figure 2.3.

On constate qu'avec ce modèle simple, il existe dans les deux cas des processus auxquels aucun agent n'est assigné, créant un déséquilibre de charge significatif. En effet, cette méthode permet d'équilibrer la charge seulement lorsque le nombre d'agents est largement supérieur au nombre de processus, et que les agents sont répartis uniformément sur l'environnement.

Cette méthode est notamment utilisée par les plateformes Repast HPC [42], D-MASON [44] et Pandora [114] décrites dans la suite. Le découpage de l'environnement est alors statique. Les travaux autour de la plateforme D-MASON prévoient l'introduction d'un découpage en bandes dynamique pour s'adapter aux distributions non uniformes

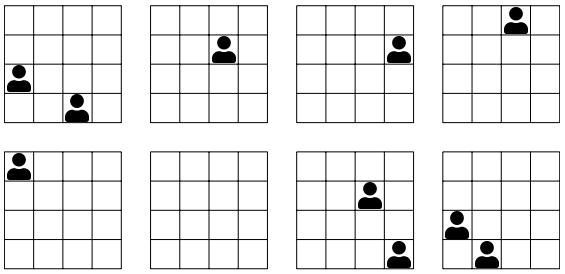


FIGURE 2.3. – Exemple de partition en grille.

d'agents dans l'environnement [44], comme étudié dans d'autres travaux [26]. La plateforme Distributed MASON [127], une autre version distribuée et expérimentale de MASON, propose quant à elle la mise en place d'un équilibrage de charge spatial à base de grille dynamique, en proposant à la fois des modifications du partitionnement à l'échelle locale et à l'échelle globale grâce à une méthode basée sur des arbres quaternaires.

Il est intéressant de noter que même si le partitionnement de l'environnement est statique, le partitionnement du modèle, qui inclut les agents, est dynamique. En effet, les agents migrent entre les processus en fonction de leur position et de l'avancée de la simulation, même si le partitionnement de l'environnement ne change pas. Il est cependant possible de mettre en place des techniques qui associent un processus à chaque agent en fonction de leur position au début de la simulation, mais ne font pas migrer les agents par la suite, indépendamment de leur position [106, 105, 125, 24].

2.7.2. Méthodes par partitionnement de graphe

Dans le domaine du Calcul Haute Performance, le problème de l'équilibrage de charge est régulièrement abordé grâce au partitionnement de graphe. Nous considérons un graphe $G = (V, E)$ où V représente les nœuds et E les liens entre eux. Un poids est associé aux nœuds et aux liens. Le problème consiste alors à décomposer l'ensemble V en k sous-ensembles disjoints en minimisant le poids total des liens reliant deux nœuds assignés à des sous-partitions différentes tout en maintenant un équilibre entre les sous-partitions en termes de poids des nœuds. Buluç et al. [32] proposent une revue moderne des algorithmes de partitionnement de graphe.

Parmi les nombreux exemples d'implémentations, nous pouvons citer PaToH [33], MeTiS [69] et Scotch [101], qui permettent d'effectuer du partitionnement de graphe de manière séquentielle. Néanmoins, dans le cadre d'une exécution distribuée large échelle, il est important que l'algorithme de partitionnement lui-même soit distribué, surtout dans le cas d'un équilibrage de charge dynamique. Ainsi les librairies Zoltan [35, 6], ParMeTiS [118] et PTScotch [38] permettent l'exécution distribuée des algorithmes de partitionnement de graphe.

Ces algorithmes peuvent permettre d'effectuer de l'équilibrage de charge dynamique de manière naïve en les appliquant au cours de la simulation pour réassigner les nœuds du graphe aux processus en fonction de l'évolution des charges et des communications. Cependant, la migration des nœuds entre les processus a elle-même un coût, et des modifications mineures d'une exécution à l'autre peuvent entrainer de nombreuses migrations inutiles. D'autre part, le gain en performance induit par la migration d'un ensemble de nœuds doit compenser le coût de la migration elle-même. Ainsi la librairie Zoltan implémente un algorithme de *repartitionnement* permettant d'implémenter un équilibrage de charge dynamique qui prend en compte le coût de migration des nœuds [34].

Blythe et Tregubov [23] et Macal et al. [82] présentent des cas d'application de la librairie MeTiS à des simulations distribuées de SMA impliquant une structure de graphe. Nos travaux se basent notamment sur une idée de Rousset et al. [113] qui consiste à représenter tout SMA grâce à un graphe afin de pouvoir effectuer l'équilibrage de charge de toute simulation grâce à Zoltan. Lui et Chan [78] proposent une solution permettant

de distribuer les avatars d'un *Environnement Virtuel Distribué* spatialisé grâce à un algorithme de partitionnement à base de graphe, dont l'efficacité dans le cadre d'une répartition non-uniforme des agents a été démontrée.

De manière générale, le passage du partitionnement de graphe à l'équilibrage de charge dans le cadre de la simulation distribuée de SMA est évident en définissant un problème de partitionnement de graphe tel que :

— k est fixé comme étant le nombre de processus ;
— les nœuds représentent les agents ;
— les poids des nœuds représentent la charge de calcul associée à chaque agent ;
— les liens représentent une interaction possible entre deux agents ;
— les poids des liens représentent le coût des communications entre deux agents.

La résolution de ce problème revient alors à minimiser la fonction de coût précédemment évoquée dans le cadre de l'équilibrage de charge.

2.7.3. Méthodes par proximité spatiale

On peut montrer que le partitionnement à base de grille offre de très bons résultats lorsque la distribution des agents sur l'environnement est uniforme.

La méthode n'est cependant pas adaptée au cas où les agents se concentrent sur certaines zones de l'environnement. C'est par exemple le cas avec des modèles de colonies de fourmis, où les fourmis sont principalement concentrées autour de la nourriture la plus proche, ou avec le modèle *Sugarscape* [53] où la nourriture n'est pas répartie uniformément sur la grille. Ces exemples se généralisent facilement à tout modèle qui pousse les agents à se rapprocher de points d'intérêt répartis de manière non uniforme sur l'environnement. L'application d'un partitionnement à base de grille sur ce type de modèle peut mener à des déséquilibres en charge de calcul significatifs entre les processus, et donc réduire les performances globales, l'avancée de la simulation étant toujours limitée par le processus le plus lent. Le problème se pose également avec les modèles où les agents ne cherchent pas un point particulier de l'environnement, mais cherchent à se rapprocher les uns des autres, comme dans les modèles de bancs de poissons ou de nuées d'oiseaux. Ces phénomènes se manifestent aussi avec les modèles de simulation de foule où les agents sont souvent amenés à se déplacer en groupe dans l'environnement vers des points d'intérêt, ou depuis des zones de danger.

Ainsi, plutôt que de partitionner la simulation en se basant sur l'environnement, Solar, Suppi et Luque [120, 121] proposent une méthode consistant à regrouper les agents en fonction de leur proximité dans le cas d'un modèle de bancs de poissons. La méthode est néanmoins généralisable à des modèles avec des agrégations d'agents similaires, par exemple dans le cas de simulations de foules.

Vigueras et al. [125] proposent et comparent trois méthodes de partitionnement dynamique qui visent plus ou moins directement à regrouper les agents d'un système spatialisé selon une relation de proximité.

Les travaux autour de la plateforme GAIA mettent en place une méthode d'équilibrage de charge dynamique dans le cas des SMA spatialisés basée sur la migration d'agents afin de réduire les communications distantes [24, 47].

2.7.4. Autres méthodes

Dans la pratique, certaines méthodes d'équilibrage de charge basiques peuvent faire leurs preuves.

Ainsi il existe des exemples d'utilisation d'un algorithme de tourniquet (« round-robin ») pour assigner les agents à chaque processus [41, 100]. Les agents de la simulation sont alors simplement stockés dans une file, et assignés à chaque processus à tour de rôle. Le temps de partitionnement de cette méthode triviale est négligeable, et permet un très bon équilibrage de la charge de calcul. En revanche, elle ne permet pas de maitriser le coût des communications. Ce problème n'est cependant pas toujours critique, par exemple dans le cas de la plateforme FLAME, où l'équilibrage et l'optimisation des communications se fait davantage au niveau des échanges de messages qu'au niveau des agents [72]. De plus, le schéma d'exécution de FLAME a rapidement été adapté pour une exécution sur GPU, pour laquelle il n'y a pas de problème de communications, la mémoire étant partagée.

2.8. Plateformes de simulation distribuée

Plusieurs plateformes permettent l'exécution distribuée de simulations de SMA, et proposent donc des solutions aux problèmes précédemment évoqués. Parmi elles, Repast HPC [42] et D-MASON [45, 44] sont des adaptations des plateformes séquentielles déjà citées. Les techniques de modélisation, par exemple pour la planification du comportement des agents ou l'environnement, sont donc sensiblement les mêmes. Les contraintes liées à la distribution introduisent cependant des limitations spécifiques. D'autres plateformes, comme FLAME [41, 108] et Pandora [114], sont nativement conçues pour le Calcul Haute Performance.

L'une des faiblesses de ces plateformes réside selon nous dans le manque de formalisme utilisé pour définir et analyser les méthodes mises en place pour résoudre les problèmes précédemment évoqués. C'est pourquoi nous proposons ici une description des plateformes qui permet de facilement les comparer en termes de distribution, de synchronisation des données et d'équilibrage de charge. Toutes utilisent la synchronisation temporelle par pas de temps.

2.8.1. Repast HPC et D-MASON

Les plateformes Repast HPC et D-MASON proposent des solutions similaires aux problèmes de synchronisation et de distribution pour les modèles spatiaux.

Équilibrage de charge

Pour les projections spatiales continues ou à base de grille, l'équilibrage est effectuée en découpant l'environnement afin d'en assigner une partie à chaque processus. Chacun est alors chargé d'exécuter tous les agents localisés dans la partie qui lui est associée. Le découpage peut-être produit automatiquement par D-MASON, mais est laissé à la

charge de l'utilisateur avec Repast HPC. Même si la distribution des projections à base de graphe est permise par ces plateformes, aucune méthode d'équilibrage de graphe n'est nativement proposée.

Distribution

La continuité des données est assurée grâce à un concept nommé « OverLapping Zones »(OLZ) ou « Areas of Interest » (AoI), que nous traduirons par *zones de recouvrement* dans la suite. Il consiste à répliquer, sur chaque processus, une portion de l'environnement associée aux processus voisins. La taille des zones peut notamment dépendre de la taille du champ de perception des agents locaux. Plus précisément, des copies des agents *distants* localisés dans les zones de recouvrement sont créées sur chaque processus. Contrairement aux agents locaux, ces copies ne sont pas exécutées par le processus local, elles permettent seulement l'accès aux données des agents *distants* par les agents *locaux*. La figure 2.4 présente un exemple de distribution sur 4 processus. L'agent *local* A_9 perçoit par exemple l'agent *local* A_8, ainsi qu'une copie de l'agent *distant* A_7.

La méthode de distribution ne se limite pas aux environnements à base de grille discrète. Pour les environnements continus, la taille des zones de recouvrement est déterminée de la même manière, en fonction de la taille des champs de perception des agents. Pour les environnements à base de graphe, la zone de recouvrement est construite à partir des voisins des agents *locaux* dans le graphe.

Synchronisation des données

Les données des agents *distants* sont importées à la fin de chaque pas de temps depuis les processus sur lesquels ils sont exécutés, la synchronisation temporelle s'effectuant par pas de temps. Les possibilités d'interactions varient cependant légèrement entre les deux plateformes.

Repast HPC autorise toute interaction entre agents *locaux* : ils peuvent modifier mutuellement leurs données sans aucune limitation. En effet, aucun problème de concurrence n'a lieu dans ce cas car les agents *locaux* sont exécutés de manière séquentielle à l'échelle d'un processus. Cependant, les interactions possibles entre agents *locaux* et *distants* sont limitées. En effet, les modifications effectuées par les agents locaux sur les agents *distants* sont systématiquement écrasées par la mise à jour en fin de pas de temps. Au sein d'un pas de temps, les données d'un agent *distant* correspondent à son état au pas de temps précédent, auquel se sont éventuellement ajoutées des modifications effectuées par les agents *locaux* pendant le pas de temps en cours, même si celles-ci seront par la suite perdues.

D-MASON propose une politique beaucoup plus stricte en termes d'interactions : les agents *locaux* ne peuvent accéder qu'à une copie *ghost* de leurs voisins, qu'ils soient *locaux* ou *distants*. Ainsi, seules les lectures sont possibles sur les voisins, mais les modalités d'accès aux agents ne dépendent pas du processus sur lequel ils sont exécutés. Chaque agent peut modifier son propre état, dont les mises à jour seront perceptibles au pas de temps suivant par ses voisins.

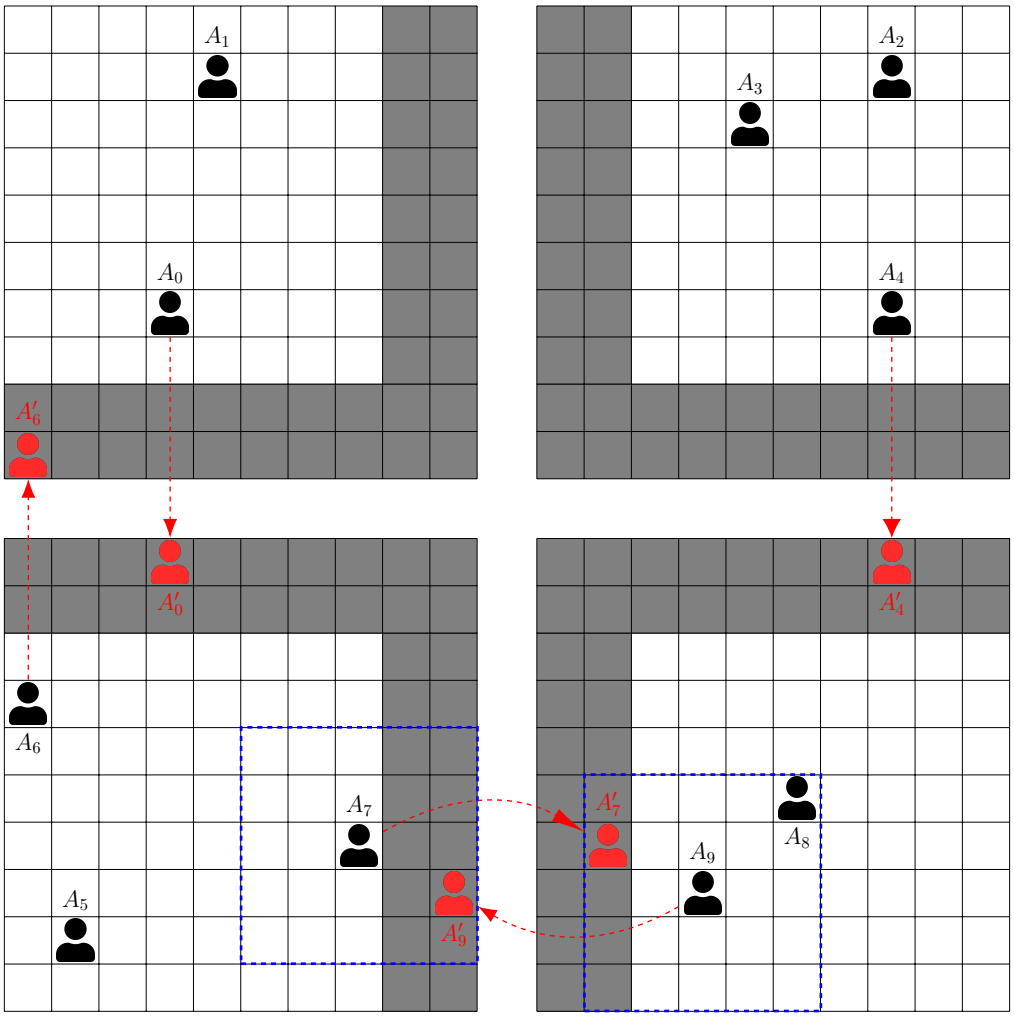


FIGURE 2.4. – Exemple de distribution sur 4 processus, chacun associé à une portion de l'environnement (cases blanches), avec représentation des zones de recouvrement (cases grises). Les agents exécutés sur chaque processus (agents noirs) peuvent interagir avec les copies des agents *distants* (représentés en rouge). Des exemples de champs de perception sont représentés en bleu.

Discussion

Contrairement à Repast HPC, la méthode de synchronisation proposée par D-MASON permet la reproductibilité des simulations, quel que soit l'ordre d'exécution des agents ou le nombre de processus utilisés.

En effet, considérons deux agents *locaux* A et B simulés avec Repast HPC, dans le cadre d'un modèle où les agents accèdent à leurs voisins en lecture seule. Si A est exécuté avant B, alors B accède à l'état de l'agent A au pas de temps courant, t. Mais si B est exécuté avant A, B accède à l'état de A au pas de temps $t-1$. En outre, si A est *distant*, B accède toujours à l'état de A au pas de temps $t-1$, quel que soit l'ordre d'exécution réel des agents. De plus, le fait que A soit *distant* ou non dépend du nombre de processus utilisés pour exécuter la simulation : plus ce nombre est élevé, moins il y a d'agents *locaux* par processus, plus il y a de voisins *distants*. Avec un raisonnement analogue, on observe que des écritures potentielles entre A et B peuvent également influer sur la reproductibilité de la simulation avec Repast HPC. Dans le cas de D-MASON, l'agent B accède toujours à l'état de l'agent A au pas de temps $t-1$, quelle que soit la situation.

Pour des raisons de gestion de l'aléatoire, l'accès aux voisins en lecture seule est cependant une condition nécessaire mais pas suffisante pour garantir la reproductibilité indépendamment du nombre de processus, comme discuté en détail dans la section 5.3.

2.8.2. Pandora

Contrairement aux exemples précédents, la plateforme Pandora [114] ne se base pas sur une plateforme séquentielle existante.

Équilibrage de charge

Comme pour les plateformes précédentes, l'équilibrage de charge proposé est basé sur le découpage de l'environnement en grille, Pandora n'étant utilisé que pour les modèles spatiaux.

Distribution

La continuité des données est assurée par des zones de recouvrement.

Synchronisation des données

Le schéma d'exécution introduit un mécanisme de synchronisation original qui permet les écritures entre les processus. En effet, chaque portion de l'environnement associée à un processus est elle-même découpée en quatre parties, comme présenté sur la figure 2.5. L'agent A_7, dans une partie bleue, peut par exemple effectuer de manière sécurisée des modifications sur la copie A'_9 de l'agent A_9 car celui-ci, localisé dans une partie verte, n'est pas exécuté en même temps que A_9. En outre, seuls des agents localisés dans la même partie que A_7 peuvent interagir avec A_7, il n'y a donc pas de problème de concurrence. La synchronisation temporelle est toujours effectuée de manière conservative, mais chaque

pas de temps est subdivisé en quatre, avec une étape de synchronisation entre chaque sous pas de temps.

Avec cette méthode, on constate que chaque sous-partie d'une couleur donnée n'est adjacente à aucune sous-partie de la même couleur. Le découpage peut être facilement adapté en fonction du nombre de processus pour conserver ces propriétés. Chaque agent peut donc modifier les données des agents des sous-parties adjacentes, qu'elles appartiennent ou non à un autre processus, car elles ne sont pas exécutées dans le sous pas de temps en cours. De plus, chaque agent dans une sous-partie non exécutée ne peut être modifié que par les agents d'un seul processus. En effet, même si chaque sous-partie P est adjacente à deux sous-parties Q_0 et Q_1 de la même couleur (par exemple, sur la figure 2.5, les parties bleues sont adjacentes à deux parties vertes), la taille des champs de perception des agents fait que les agents des parties Q_0 et Q_1 ont chacun accès à des sous-ensembles disjoints des agents de P [4] Ainsi les modifications des copies des agents dans les zones de recouvrement peuvent être transmises de manière sécurisée aux processus distants à la fin de chaque sous pas de temps. Les agents de chaque partie étant exécutés de manière séquentielle, il n'y a pas de problème d'accès concurrents aux copies [5]. Il n'y a donc pas de problème de concurrence sur toute l'exécution du pas de temps, même en autorisant la modification des agents *distants*.

Discussion

Cette méthode possède néanmoins des inconvénients notables :

1. Le nombre de barrières de synchronisation est multiplié par quatre, ce qui peut avoir un impact significatif en termes de performances.
2. La décomposition en sous pas de temps peut introduire un biais dans l'ordre d'exécution des agents.
3. La méthode ne peut s'appliquer qu'à des modèles à base de grille régulière, et limite les interactions des agents à leur champ de perception géographique (pas de support pour un graphe de contacts par exemple). Dans le cas d'un environnement à base de graphe, il serait cependant intéressant de réfléchir à une méthode similaire basée sur les problèmes de coloration de graphes.

2.8.3. FLAME

La plateforme FLAME [41] et son adaptation aux architectures GPU [108] introduisent d'autres concepts permettant la distribution des simulations de SMA. Cette plateforme a également été conçue spécifiquement pour l'exécution distribuée, grâce à une technique de modélisation permettant la parallélisation native des simulations. Actuellement, le projet FLAME GPU est activement en développement, contrairement à sa version originale

4. En considérant que la taille des sous-parties est suffisamment grande par rapport à la taille des champs de perception, ce qui ne pose pas de problème dans le cadre d'exécutions large échelle.

5. Même dans le cas d'une exécution parallèle des agents des sous-parties, seuls des problèmes de concurrence en mémoire partagée se posent.

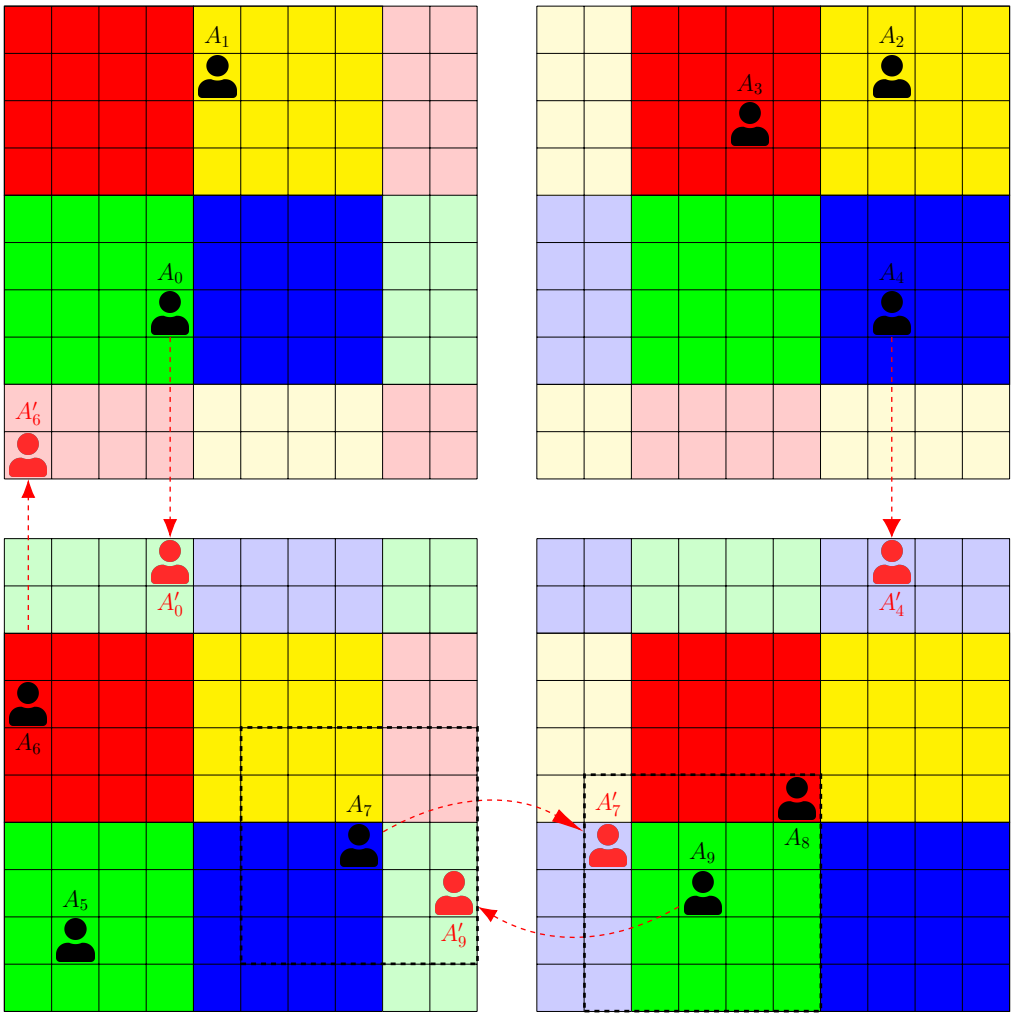


FIGURE 2.5. – La distribution d'un modèle selon Pandora. Les zones claires représentent les zones de recouvrement. Les parties de l'environnement de même couleur sont exécutées simultanément, avec à chaque étape la garantie qu'aucune zone d'une autre couleur n'est exécutée sur d'autres processus.

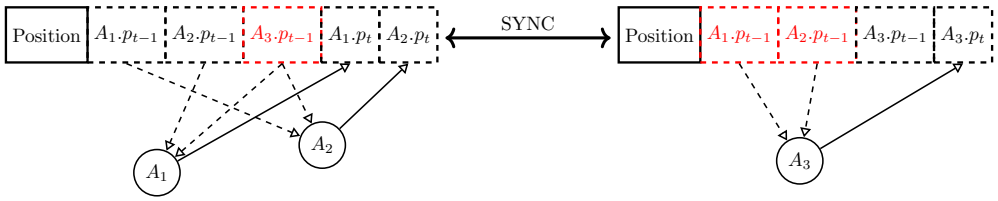


FIGURE 2.6. – Exemple d'utilisation d'un tableau de messages avec 3 agents et 2 processus.

sur CPU. Même si l'exécution GPU n'entre pas dans le cadre de notre étude, les deux plateformes sont basées sur les mêmes concepts.

Les agents sont décrits comme des machines à état : à chaque instant, chaque agent se trouve dans un état particulier parmi un nombre fini d'états possibles. Des fonctions implémentées par l'utilisateur permettent de décrire les transitions entre états.

Distribution

D'après les auteurs de FLAME [41], l'accès direct d'un agent aux données des autres rend difficile la conception de plateformes de simulation distribuées, surtout partant d'une plateforme séquentielle existante qui n'a pas à considérer ce genre de problèmes. Dès lors, une solution qui s'affranchit totalement des accès mutuels aux données des agents a été proposée. Dans FLAME, les agents ne peuvent échanger des données que par le biais de messages explicites transmis via un tableau de messages. Chaque agent peut publier ou s'inscrire à un type de message sur un tableau afin de réagir aux actions de ses voisins au cours de l'exécution de ses fonctions de transitions. L'exécution des agents est nativement parallèle car il n'existe pas de dépendances directes entre les agents. La souscription à un type de messages permet ici d'assurer la continuité des données. L'exemple de la figure 2.6 présente un exemple de partage de la position des agents. Les agents ne peuvent accéder directement à la position des autres, mais ils publient à chaque pas de temps leur position actuelle dans le tableau de messages (messages $A_i.p_t$), et ils peuvent lire pendant le pas de temps la position des autres agents publiée au pas de temps précédent (message $A_i.p_{t-1}$). Il est possible d'ajouter autant de propriétés que nécessaire au tableau de messages.

Synchronisation des données

La synchronisation des tableaux de messages consiste en l'échange des messages entre les processus et a lieu à la fin de chaque pas de temps, grâce à des communications collectives.

Équilibrage de charge

La distribution des agents sur les processus peut se faire selon la position des agents, ou selon un algorithme de tourniquet.

Le processus de synchronisation optimise en outre les échanges d'informations en ne transmettant pas les informations auxquelles aucun agent n'est inscrit, ou grâce à des filtres spécifiés par l'utilisateur. Cela permet par exemple de recevoir seulement les attributs des agents localisés dans le champ de perception d'au moins un agent *local*. L'envoi de messages précis permet de minimiser davantage les transferts d'informations, là où les méthodes par zones de recouvrement copient généralement l'intégralité des données des agents à chaque pas de temps.

Les lectures et écritures dans les tableaux de messages et les fonctions de transitions qui les effectuent étant spécifiées par l'utilisateur, la plateforme FLAME cherche enfin à optimiser automatiquement l'ordre d'exécution des agents lors de la compilation du modèle afin de maximiser le recouvrement calcul / communications.

Discussion

FLAME introduit un formalisme de description des modèles basé sur des fichiers XML et l'implémentation de fonctions de transition en C. La génération de code et la gestion de l'exécution distribuée sont ensuite automatiquement gérées par la plateforme : l'utilisateur n'a donc pas besoin de compétences en parallélisme, et la modélisation des systèmes simulés est indépendante de l'exécution distribuée.

Cette méthode peut cependant restreindre les modèles simulés : dans certains cas, l'accès direct aux données des agents voisins ne peut être contourné grâce aux tableaux de messages sans modifier les règles du modèle d'origine, comme présenté précédemment avec le problème du vol de ressource. La famille de techniques de modélisation dites « Influence / Reaction » décrites précédemment à la section 2.2 semble cependant particulièrement compatible avec l'utilisation des tableaux de messages.

2.8.4. Autres Travaux

La liste de plateformes proposée ici n'est pas exhaustive, et se focalise sur les concepts pertinents dans le cadre de notre propre réflexion. Une revue plus complète des plateformes de simulation distribuée a été proposée en 2016 par Rousset et al. [112].

Il existe néanmoins d'autres projets dignes d'intérêt, même s'ils n'aboutissent pas nécessairement à une plateforme de simulation générique accessible au grand public.

Ainsi le projet CareHPS [25] propose une plateforme de simulation distribuée de SMA qui se focalise sur la modularité de ses composants. En effet, les exemples précédents sont relativement rigides et peu extensibles en termes de distribution, d'équilibrage de charge et de synchronisation des données. CareHPS définit une architecture basée sur des interfaces génériques, ce qui permet de facilement implémenter de nouveaux algorithmes d'équilibrage de charge ou de synchronisation afin de s'adapter au mieux au SMA simulé. Les sources de ce projet ne sont cependant pas disponibles au moment de la rédaction.

Le projet FARM [23], dédié à la simulation distribuée de réseaux sociaux, propose une méthode d'équilibrage de charge basée sur le partitionnement de graphe réalisé à l'aide de METIS [70]. Les interactions entre agents sont permises grâce à la solution Apache ZooKeeper. Cette solution, en partie centralisée, ne semble cependant pas adaptée à la quantité d'interactions qui peut avoir lieu entre les agents au cours d'une simulation.

Les travaux de Rousset et al. [113] précédant ce projet proposent la mise en place de divers *modes de synchronisation* dont certains permettent les écritures distantes et concurrentes au cours du pas de temps, mettant en évidence l'impact des modes de synchronisation sur les performances et les résultats des modèles.

2.9. Synthèse

Cette étude est focalisée sur la simulation distribuée de SMA génériques, sans contrainte sur la structure de l'environnement et des agents, exécutée sur des architectures de Calcul Haute Performance distribuées et homogènes. Nos travaux ainsi que l'étude de la littérature nous ont permis d'identifier des problèmes génériques auxquels toute simulation distribuée de SMA doit faire face : les mécanismes de distribution de l'exécution et de continuité des données, la synchronisation des données, la synchronisation temporelle et l'équilibrage de charge. L'étude de l'état de l'art et des plateformes existantes permettent d'apporter des solutions à ces problèmes, issues de la simulation de SMA ou du calcul distribué de manière générale. Même si les analyses théoriques et pratiques de ces problèmes ne sont pas systématiquement mentionnées dans la conception des plateformes existantes, notre revue nous a permis d'identifier clairement quelles solutions leur sont apportées par chaque plateforme, ce qui permet de valider l'aspect générique de ces problèmes et de mettre en valeur la diversité des techniques de simulation distribuée ainsi que leurs avantages et inconvénients. Dans la seconde partie, nous proposons une analyse approfondie des problèmes liés à la distribution ainsi qu'une architecture logicielle générique permettant de les résoudre de manière flexible. La synchronisation temporelle étant systématiquement réalisée par pas de temps, nous ne reviendrons pas sur ce point. Les mécanismes de distribution, l'équilibrage de charge et la synchronisation des données sont cependant respectivement abordés dans les chapitre 3, 4 et 5.

Deuxième partie

Conception et analyse d'une architecture logicielle générique dédiée à la simulation distribuée de Systèmes Multi-Agents

Dans la première partie, nous avons montré que toutes les tentatives de distribution de simulations de SMA se confrontent à de nombreux problèmes communs tels que la continuité des données, l'équilibrage de la charge, la synchronisation temporelle ou la synchronisation des données. La diversité des plateformes proposées prouve l'absence de consensus quant aux techniques à utiliser pour les résoudre. Les différentes méthodes rencontrées sont cependant rarement confrontées entre elles, de manière théorique ou expérimentale, et les choix d'implémentation ne sont pas toujours justifiés.

Dans le domaine de la simulation Multi-Agents, on constate en outre que les besoins varient indépendamment des techniques de modélisation ou des choix d'implémentation des plateformes. Dans certains cas, il est par exemple souhaitable d'effectuer des simulations parfaitement reproductibles, dans d'autres on privilégiera la liberté d'interaction entres les agents, et dans d'autres encore on cherchera avant tout à minimiser le temps d'exécution de la simulation. Les exemples étudiés montrent qu'il est difficile voire impossible de mettre en place des solutions uniques pouvant satisfaire simultanément la grande diversité de besoins utilisateurs. Pourtant, concevoir une plateforme capable de simuler tout type de modèle semble réaliste et souhaitable. Atteindre cet objectif va donc nécessiter de concevoir une plateforme modulable, permettant à l'utilisateur de choisir les solutions les plus adaptées à ses besoins, voire d'en mettre en place de nouvelles.

On constate que la tendance générale consiste à laisser à l'utilisateur la charge d'adapter ses modèles à la plateforme de simulation choisie. Or il est parfois impossible d'adapter certains modèles sans modifier radicalement la définition du comportement des agents, en raison de trop fortes contraintes imposées par les plateformes de simulation. Une contrainte évidente est notamment l'impossibilité de réaliser des écritures concurrentes entre des agents exécutés sur différents processus, alors que la définition des modèles repose régulièrement sur ce type d'interaction, trivial à mettre en place dans le cadre d'une exécution séquentielle. Les algorithmes d'équilibrage de charge proposés ne s'adaptent également pas toujours à tous les modèles, par incompatibilité avec le modèle simulé ou pour des raisons de performances.

Enfin, les interfaces proposées requièrent régulièrement de la part de l'utilisateur une connaissance des enjeux et méthodes du calcul distribué, ce qui peut sévèrement limiter l'utilisation des ressources de Calcul Haute Performance par la communauté Multi-Agents. Ces interfaces ne sont de fait pas toujours adaptées aux habitudes des utilisateurs avec les plateformes non distribuées.

La faible extensibilité des plateformes de simulation distribuée existantes ne permet donc pas de fournir à l'utilisateur final un niveau d'abstraction et de flexibilité suffisant, ni de mettre en place nos expérimentations pour analyser les problèmes liés à la distribution des simulations de SMA.

D'où notre motivation à concevoir notre propre plateforme de simulation distribuée de SMA, FPMAS [1], dans la continuité des travaux d'Alban Rousset [111]. Notre contribution ne se limite cependant pas seulement à une plateforme de plus, qui prétendrait résoudre de manière pérenne les limitations de ses homologues. En effet, en plus des fonctionnalités inédites et des comparaisons entre algorithmes d'équilibrage de charge ou de synchronisation de données permises par la plateforme, nous prétendons, au travers du retour d'expérience du développement de FPMAS, décrire formellement

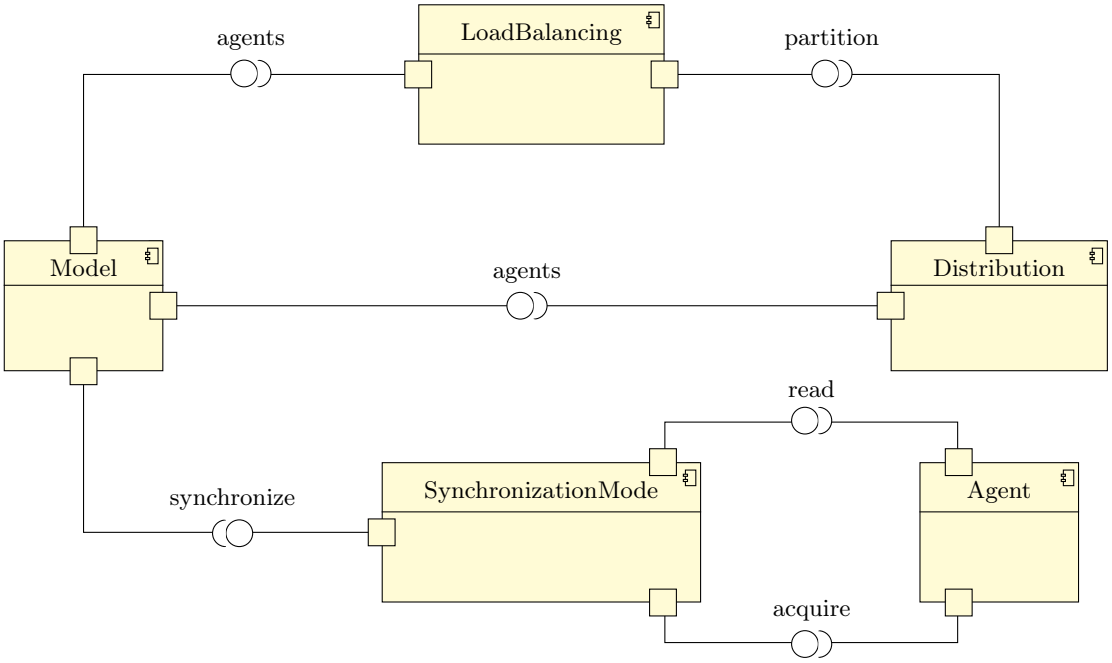


FIGURE 2.7. – Diagramme de composants de l'architecture logicielle générique proposée.

les problèmes et solutions liés à toute conception d'outils pour l'exécution distribuée de simulation de SMA, ainsi que des algorithmes génériques adaptables à d'autres travaux. Nos contributions peuvent finalement se résumer ainsi :

1. Définitions formelles des problèmes rencontrés au cours de l'expérience de développement de FPMAS, et généralisation à la simulation distribuée de SMA dans un environnement matériel et logiciel arbitraire.
2. Conception d'une architecture logicielle générique grâce à une décomposition en interfaces à implémenter, qui constituent un cahier des charges des fonctionnalités à mettre en place pour distribuer une simulation Multi-Agents, ou pour concevoir une plateforme de simulation distribuée de SMA.
3. Propositions d'implémentation des interfaces proposées, à la fois dans notre plateforme FPMAS, et sous forme d'algorithmes génériques afin de permettre une utilisation dans d'autres contextes.
4. Lorsque cela est pertinent pour l'utilisateur, comparaisons théoriques et expérimentales des différentes solutions proposées, afin de permettre aux utilisateurs et aux développeurs de choisir et de mettre en place les solutions adaptées à leurs besoins grâce à l'aspect modulable intrinsèque de l'architecture proposée.

L'architecture logicielle proposée est représentée sur la figure 2.7, et détaillée dans les chapitres suivants.

Chapitre 3.

Distribution des Systèmes Multi-Agents

Nous commençons la description de notre architecture logicielle de distribution des simulations de SMA par des problèmes génériques et fondamentaux liés à la distribution. Nous définissons d'abord le contexte général de l'étude, d'une part en termes de parallélisme dans la section 3.1, et d'autre part en termes de SMA dans la section 3.2. Dans la section 3.3, nous introduisons des algorithmes permettant de mettre en place une distribution cohérente et arbitraire de tout SMA sur une architecture de calcul distribuée, notamment pour mettre en place l'exécution des agents *locaux* sur chaque processus et assurer la continuité des données grâce aux agents *délégués*. Dans la section 3.4, nous présentons le cas particulier de la représentation à base de graphe des SMA utilisée par FPMAS, qui permet quelques simplifications des algorithmes précédents. Pour finir, nous abordons brièvement dans la section 3.5 d'autres problèmes marginaux liés à la simulation distribuée de manière générale, comme la sérialisation des données ou la génération de nombres aléatoires.

Table des matières

3.1. Contexte d'exécution distribuée

De manière générale, la parallélisation d'un programme consiste à le décomposer en un ensemble de *processus* à exécuter en parallèle. Chaque processus est exécuté sous forme d'une ou plusieurs *tâches* (« threads ») par un ensemble de *processeurs*. Pour simplifier, nous considérons qu'un processeur correspond à une seule unité d'exécution, c'est-à-dire à un coeur de processeur dans le sens commun. Nous qualifions alors un système physique de *multiprocesseur* lorsqu'il dispose de plusieurs processeurs, comme c'est le cas pour la plupart des architectures modernes. Nous définissons un système d'exploitation *multitâche* comme un système capable d'exécuter plusieurs tâches sur le même processeur. Ainsi l'exécution d'un processus n'est pas nécessairement liée à un unique processeur et inversement : un processus peut être exécuté sur plusieurs processeurs, et un processeur peut exécuter plusieurs processus.

Dans un système en mémoire *partagée*, tous les processeurs ont un accès direct à une mémoire commune. Un système *distribué* est constitué d'un ensemble de machines en mémoire partagée connectées en réseau, appelées *nœuds de calcul*. Chaque nœud possède un ensemble de processeurs. Les processeurs d'un nœud ne partagent donc pas de mémoire avec les autres nœuds. L'exécution distribuée d'un programme consiste ainsi à le décomposer en un ensemble de processus à exécuter sur un système distribué. Chaque nœud de calcul peut héberger plusieurs processus, mais un processus ne peut être assigné qu'à un seul nœud. Nous travaillons sous l'hypothèse du principe d'isolation, généralement mis en place au niveau du système d'exploitation, selon lequel les différents processus ne peuvent accéder directement aux espaces mémoire alloués aux autres, contrairement aux tâches associées à un même processus qui peuvent partager leur espace mémoire.

La figure 3.1 présente un exemple de décomposition de simulation en huit processus exécutés sur deux nœuds de calcul. Chaque cœur de processeur est chargé de l'exécution d'un seul processus, associé à une seule tâche. Chaque tâche ne peut accéder à la mémoire de travail des autres, y compris sur le nœud local, car elles sont assignées à d'autres processus.

La figure 3.2 présente quant à elle la décomposition de la même simulation en deux processus, sur la même architecture. Cette fois, plusieurs tâches sont assignées à l'exécution d'un processus. Dans le cas d'une simulation Multi-Agents, cela signifie que les agents exécutés par une tâche auront un accès direct aux agents exécutés par les autres tâches associées au même nœud de calcul afin de mettre en place une exécution parallèle en mémoire partagée, ce qui n'est pas possible avec le schéma d'exécution de la figure 3.1

Pour des raisons de simplicité, nous ne nous intéressons pas dans le cadre de cette étude à la décomposition des processus en tâches, c'est-à-dire au schéma d'exécution des processus. Nous nous focalisons seulement sur la décomposition d'une simulation de SMA en un ensemble de processus distribués, indépendamment de l'architecture du système chargé d'exécuter la simulation. A ce niveau d'abstraction, la seule différence entre les figures 3.1 et 3.2 est le nombre de processus utilisés pour décomposer la simulation. Dès lors, les solutions proposées, qui consisteront à distribuer la simulation sur un nombre

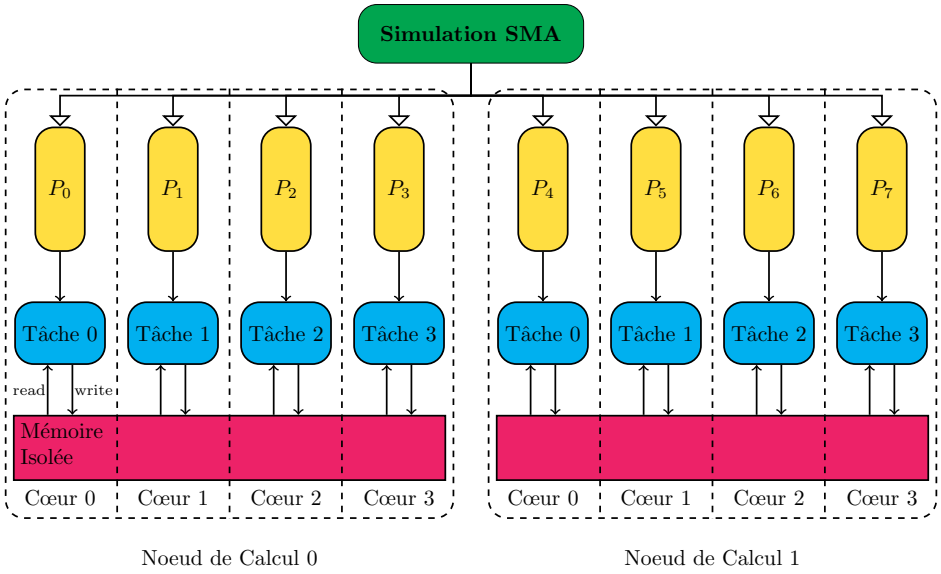


FIGURE 3.1. – Exemple de décomposition d'une simulation SMA en 8 processus, exécutés en tâche simple sur 2 nœuds de calcul avec chacun 4 processeurs.

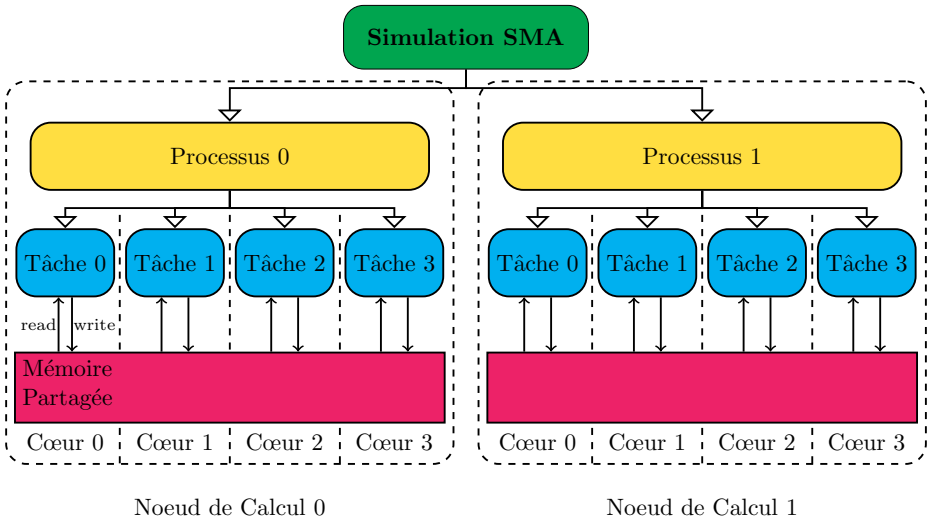


FIGURE 3.2. – Exemple de décomposition d'une simulation SMA en 2 processus, exécutés en multitâche sur 2 nœuds de calcul avec chacun 4 processeurs.

arbitraire de processus, pourront s'appliquer indépendamment du schéma d'exécution ou de l'architecture matérielle.

3.2. Contexte Multi-Agents

Notre objectif est de concevoir des méthodes qui permettent la distribution de la plus large catégorie de SMA possible. Nous imposons cependant quelques contraintes aux SMA tels que définis dans la section 2.1 afin de définir le contexte de l'étude. Les solutions proposées peuvent éventuellement s'appliquer dans d'autres contextes, même s'ils n'ont pas été abordés.

3.2.1. Exécution par pas de temps

Nous restreignons notre étude aux modèles dont le schéma d'exécution est décrit par pas de temps, en accord avec la spécification de nombreux modèles existants. Les comportements des agents s'exécutent alors à des dates définies, à intervalles réguliers ou non, en opposition à la modélisation par évènements où l'exécution des comportements est déclenchée en réponse à des évènements émis par l'environnement ou par d'autres agents lors de l'exécution de leurs propres comportements.

Le schéma d'exécution par pas de temps n'implique ni l'exécution de tous les agents à chaque pas de temps, ni la régularité de l'exécution de chaque comportement. Le schéma d'exécution peut être dynamique : il est alors possible de planifier, depuis un comportement exécuté à une date t, l'exécution d'un comportement à toute date $t + \Delta t$, avec $\Delta t > 0$. Du point de vue de la modélisation, des comportements planifiés à la même date sont supposés s'exécuter simultanément. La plateforme d'exécution peut cependant les exécuter au moins en partie de manière séquentielle, mais l'ordre d'exécution n'est alors pas spécifié, et de préférence aléatoire. Dans tous les cas, le principe de causalité s'applique : un comportement ne peut être exécuté qu'à partir du moment où tous les comportements planifiés à une date strictement antérieure ont déjà été exécutés.

Ainsi, dans le cadre de la simulation distribuée, nous imposons l'utilisation d'une synchronisation temporelle par pas de temps, mise en place par des barrières de synchronisation strictes.

3.2.2. Environnement

La représentation de l'environnement dans les SMA est un problème complexe, comme l'atteste la diversité des approches recensées dans la littérature. Certains modèles, basés uniquement sur les interactions entre agents, n'ont pas besoin d'un quelconque environnement. Dans la plupart des cas, l'environnement est décrit comme un espace dans lequel les agents peuvent se déplacer pour percevoir et interagir avec des objets inertes ou d'autres agents.

Le rôle de l'environnement se limite ainsi parfois à assigner aux agents une position dans l'espace. D'où les concepts de « projection » ou de « champ » utilisés respectivement par les plateforme Repast et MASON. Des exemples de projections sont présentés

Projection	Position
Espace 2D discret	Coordonnées 2D discrètes
Espace 3D discret	Coordonnées 3D discrètes
Espace 2D continu	Coordonnées 2D continues
Espace 3D continu	Coordonnées 3D continues
GIS (Geographical Information System)	Coordonnées Géographiques
Graphe	Nœud d'un graphe

TABLE 3.1. – Exemple de projections et types de position associés.

dans le tableau 3.1. Dans ce cas, l'environnement ne contient pas directement des objets avec lesquels les agents peuvent interagir. La position des agents est parfois suffisante pour permettre les interactions, par exemple pour transmettre un virus ou une rumeur. L'environnement est cependant souvent constitué d'entités matérielles passives ou actives : un obstacle, de la nourriture, de l'herbe qui croît... Par exemple, pour couvrir l'environnement d'herbe, on peut définir un type d'agent herbe, associée à une projection 2D discrète, puis positionner une instance d'agent herbe sur chaque coordonnée. La projection permet alors, pour un herbivore localisé en (x, y) sur la même projection, de retrouver l'agent herbe aux coordonnées (x, y). De manière générale, il est possible d'assimiler les données d'une « projection » à une donnée de localisation associée à chaque agent. Nous assimilons ainsi la distribution d'un environnement abstrait de type « projection » à la distribution des données des agents.

Les « projections » peuvent par ailleurs définir des algorithmes de requêtes spatiales, par exemple pour retrouver les plus proches voisins d'un agent. Dans la plupart des cas, l'adaptation de ces algorithmes depuis un environnement en mémoire partagée vers un environnement distribué peut s'effectuer de manière triviale grâce au maintien de la continuité des données entre les processus. Par exemple, la projection 2D discrète peut permettre de retrouver tous les voisins d'un agent dans son voisinage de Moore. Or, en distribué, le maintien de la continuité des données doit assurer l'accessibilité de ces agents depuis le processus local. L'algorithme de requête des voisins en mémoire partagée peut donc trivialement s'appliquer au contexte distribué. La mise en place d'algorithmes ou de structures de données plus complexes pour optimiser les requêtes dans un environnement distribué pourrait être envisagée, mais n'est pas abordée dans ces travaux.

Dans le cas de l'utilisation d'une projection, comme dans Repast ou Mason, les objets de l'environnement sont assimilés à des agents. Dans d'autres cas, l'environnement est représenté par une structure distincte des agents. Ainsi, dans NetLogo, les cellules de l'environnement 2D discret sont représentées par des « patchs », auxquels il est possible d'associer un comportement. Dans GAMA, une « espèce » spéciale d'agent, « grid », est utilisée de façon similaire. Dans notre contexte de simulation distribué, il est nécessaire de mettre en place une méthode de distribution de l'environnement. Or les interactions entre un agent et une partie de l'environnement, intrinsèquement associée à des données, sont similaires aux interactions entre agents.

Dans la suite de notre étude, nous faisons donc le choix d'assimiler tous les objets

de l'environnement à un type d'agent dans la simulation, auquel aucun comportement n'est éventuellement associé dans le cas d'un environnement passif. Afin de reproduire le mécanisme des projections, il suffit d'inclure une position explicite à l'état des agents.

Cette représentation permet de grandement simplifier la résolution des problèmes de distribution. En effet, les algorithmes mis en place pour distribuer des agents sont nativement applicables à l'environnement. Ce choix n'est pas pour autant limitant. La plupart des algorithmes spécifiés en termes d'agents seraient également applicables aux parties de l'environnement dans un contexte de simulation où un environnement serait représenté indépendamment du concept d'agent. Nous supposons cependant que l'environnement est décomposable en sous-parties, sans quoi la distribution de l'environnement n'aurait que peu de sens.

3.3. Algorithmes de distribution génériques

L'étude des plateformes et simulations distribuées existantes nous a montré que toutes proposent des solutions au problème de la distribution des modèles, avec diverses techniques de maintien de la continuité des données. Nous proposons ici une formalisation de ce problème ainsi que des algorithmes génériques permettant de le résoudre.

Pour rappel, la décomposition d'une simulation de SMA en processus parallèles consiste à assigner l'exécution d'une sous-partie des agents à chaque processus disponible selon une *partition*. Afin d'assurer la continuité des données, on peut ajouter aux agents *locaux* des agents *délégués*, qui représentent et permettent l'accès aux agents *distants*, sans spécifier les modalités d'interactions avec eux pour le moment. La *localisation* d'un agent correspond au processus qui l'exécute. Pour les agents *locaux*, il s'agit du processus en cours. La localisation d'un agent *délégué* correspond au processus qui exécute l'agent *distant* qu'il représente.

Dans un contexte distribué, il est nécessaire d'associer à chaque agent un *identifiant* unique à l'échelle globale de la simulation, et pas seulement à l'échelle d'un processus. Des méthodes contextuelles peuvent être mises en place, par exemple grâce à un système d'identifiants externes, tels que les identifiants des nœuds d'un graphe OpenStreetMap. Néanmoins, pour supporter le cas générique, FPMAS, tout comme Repast HPC, utilise une paire constituée de l'identifiant du processus sur lequel a été créé l'agent et d'un entier incrémenté au fur et à mesure de la création d'agents sur ce processus. L'identifiant d'un agent contient ainsi une référence vers son processus d'origine, et reste constant tout au long de la simulation, y compris en cas de migrations de l'agent vers d'autres processus.

Même s'il ne sont pas systématiquement formalisés, ces concepts d'agents *locaux*, *distants* ou *délégués*, de *partitionnement*, de *localisation* et d'*identifiant* constituent plus ou moins directement une base commune à tous les exemples de distributions de simulation de SMA recensés dans la littérature. Les problématiques associées, comme la gestion des agents *délégués* ou la mise à jour de la *localisation* des agents, sont alors communes à toute tentative de distribution d'un SMA. Nous définissons donc un composant logiciel `Distribution` comme sur la figure 3.3, chargé de distribuer les agents d'un modèle tout en maintenant sa continuité et sa cohérence.

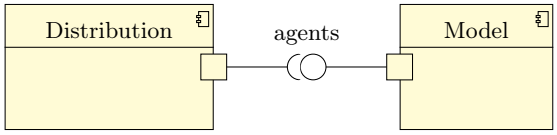


FIGURE 3.3. – Diagramme de composants UML pour l'interface de distribution.

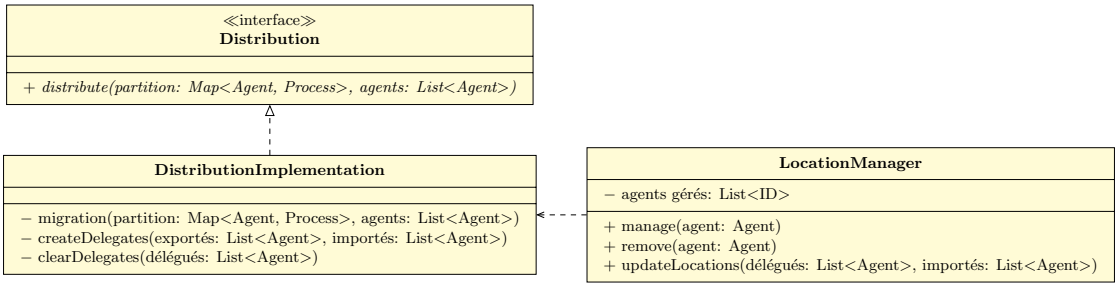


FIGURE 3.4. – Diagramme de classe de l'interface de distribution et de l'implémentation proposée.

Il est possible de formaliser des algorithmes génériques basés uniquement sur les seuls concepts précédemment introduits, avec un haut niveau d'abstraction. Les solutions proposées pourront donc toujours être implémentées, pour tout modèle Multi-Agents, quel que soit l'environnement matériel ou logiciel, même s'il est possible d'imaginer d'autres solutions spécifiques et optimisées prenant en compte la spécificité de certains modèles. Un exemple de mise en place de distribution de simulation de SMA basée sur ces algorithmes dans le cadre de l'utilisation d'une structure de données à base de graphe pour représenter les SMA est notamment détaillée dans la section 3.4.

Afin de mettre en place une solution générique au problème de la distribution, nous proposons l'implémentation de l'interface de `Distribution` présentée dans la figure 3.4. L'interface utilisateur prend en paramètre une partition arbitraire, qui associe chaque agent à un processus, et la liste d'agents à distribuer selon la partition. Notre implémentation met en place la distribution en quatre étapes :

1. `migration()` : migration des agents *locaux* vers les processus qui leur ont été assignés ;
2. `createDelegates()` : création des agents *délégués* nécessaires pour maintenir la continuité des données ;
3. `clearDelegates()` : suppression des agents *délégués* inutiles ;
4. `updateLocations()` : mise à jour de la localisation des agents *délégués*.

Nous détaillons dans la suite l'implémentation des ces méthodes sous forme d'algorithmes génériques.

3.3.1. Migration

La migration d'un agent consiste à assigner son exécution à un autre processus pour mettre en place un nouveau partitionnement. Cette opération va entre autre nécessiter d'exporter les données d'agents *locaux* vers d'autres processus. Les processus étant distribués, l'exportation ne peut se faire que par l'envoi de messages explicites : l'envoi et la réception d'un agent va donc nécessiter un processus de sérialisation, abordé dans la section 3.5.1.

Algorithme 1 Migration des agents *locaux*.

Entrée:

1: Agents *locaux* à distribuer
2: Nouvelle Partition

Sortie:

3: Agents *locaux* distribués selon la nouvelle partition

```
4: algorithme MIGRATION
5:     pour chaque (agent local, processus) de la nouvelle partition faire
6:         si processus ≠ processus actuel alors
7:             ENVOYER À processus : agent local
8:             Transformer l'agent local en agent délégué
9:         fin si
10:    fin pour
11:    RECEVOIR DEPUIS tous les processus : agents importés
12:    pour chaque agent des agents importés faire
13:        si un agent délégué représente l'agent alors
14:            Transformer l'agent délégué en agent local
15:        sinon
16:            Ajouter l'agent en tant que nouvel agent local
17:        fin si
18:        Mettre à jour la localisation de l'agent local avec le processus en cours
19:    fin pour
20: fin algorithme
```

L'algorithme 1 présente un algorithme de migration générique. L'objectif consiste, à partir du partitionnement actuel, à migrer les agents *locaux* pour obtenir en sortie d'algorithme une distribution des agents *locaux* correspondant à la nouvelle partition.

L'algorithme s'exécute simultanément sur tous les processus. La boucle ligne 5 garantit que chaque processus ne travaille que sur ses agents *locaux*. Il n'est donc nécessaire de fournir en entrée de l'algorithme que la sous-partie de la partition contenant des entrées relatives aux agents *locaux*, d'où l'aspect distribué de l'algorithme. Les localisations des agents non spécifiés dans la partition sont supposées inchangées. Chaque agent local est envoyé, si nécessaire, au nouveau processus auquel il est assigné (ligne 7). Dans un premier temps, l'agent exporté est remplacé par un agent *délégué* (ligne 8) car son accès

peut être requis par des agents *locaux* à l'issu de la migration. Le nettoyage des agents délégués inutiles est abordé dans la suite. Les agents reçus (ligne 11) sont ensuite intégrés au processus local. Si un agent *délégué* représente déjà l'agent importé sur le processus, il est remplacé par le nouvel agent *local* (ligne 14). Sinon, il suffit d'ajouter un nouvel agent *local* au processus (ligne 16). Pour finir, la localisation du nouvel agent *local* est mise à jour avec le processus en cours (ligne 18).

Une fois les agents *locaux* en place, la création d'agents *délégués* est nécessaire pour assurer la continuité des données.

3.3.2. Création et nettoyage des agents délégués

Le maintien de la continuité des données grâce aux agents *délégués* permet d'abstraire l'aspect distribué de la simulation du point de vue des agents. En effet, les agents *locaux* ou *délégués* perçus par un agent *local* seront les mêmes que dans un environnement en mémoire partagée.

Nous définissons le *voisinage* d'un agent comme l'ensemble des agents avec qui il peut être amené à interagir au cours du prochain pas de temps. Ces voisins contiennent par exemple les perceptions d'un agent, définies selon les règles du modèle. Dans le cadre de la représentation de l'environnement par des agents, les voisins peuvent représenter des parties de l'environnement vers lesquelles un agent peut se déplacer ou avec lesquelles il peut interagir. D'autres notions de voisinage sont envisageables. Dans tous les cas, les voisins peuvent être représentés par des agents *locaux* ou *délégués*. En termes d'exécution distribuée, l'accès à un agent *local* se fait par des accès mémoires directs, et l'accès à un agent *distant* se fait par des communications au travers des agents *délégués*. Nous verrons dans le chapitre 5, qui traite de la synchronisation des données, qu'il est possible d'abstraire complètement ces nuances du point de vue de l'utilisateur, afin d'assurer la transparence de la distribution.

Par définition, à chaque pas de temps, les agents *locaux* ne peuvent interagir qu'avec les autres agents *locaux* et les agents *délégués* sur le processus en cours. Afin d'assurer la continuité des données, il est donc nécessaire de construire l'ensemble des agents *délégués* de sorte qu'il contienne le voisinage de tous les agents *locaux*.

Dès lors, en supposant les ensembles d'agents *délégués* correctement initialisés, la création d'agents *délégués* pour maintenir la continuité des données peut avoir lieu dans un nombre limité de contextes :

1. Suite à la migration d'agents *locaux* pour mettre en place une nouvelle partition du modèle ;
2. Suite à un changement d'état des agents ou de l'environnement lors de l'exécution d'un comportement [1] ;
3. Suite à la création ou à la suppression d'agents.

1. Le déplacement d'un agent (au sens géographique) peut par exemple nécessiter de compléter le voisinage d'un agent avec de nouveaux agents *délégués* localisés dans son champs de perception pour respecter les règles du modèle.

Pour des raisons de simplicité, seul le premier cas est abordé dans ce document. Les deux autres dépendent des types de modèle et du comportement des agents, là où le premier, plus fondamental, s'applique à toute simulation de SMA à chaque distribution du modèle.

L'algorithme 2 présente ainsi un algorithme générique permettant d'exporter et de reconstruire le voisinage des agents importés dans le cadre d'un changement de distribution.

Algorithme 2 Création des agents *délégués*.

Entrée:

1: Agents exportés avec leur voisinage à jour
2: Agents importés sans voisinage

Sortie:

3: Agents importés avec leur voisinage à jour

```
4: algorithme CreateDelegates
5:     pour chaque agent des agents exportés faire
6:         envoyer à nouveau processus de l'agent:
7:             • voisinage de l'agent
8:             • localisation des agents du voisinage
9:         fin
10:    fin pour
11:    recevoir depuis tous les processus : voisinages des agents importés
12:    pour chaque voisinage faire
13:        pour chaque agent du voisinage faire
14:            si aucun agent local ou délégué ne représente l'agent alors
15:                Créer un nouvel agent délégué
16:                Initialiser la localisation de l'agent délégué
17:            fin si
18:            Mettre à jour le voisinage de l'agent importé
19:        fin pour
20:    fin pour
21: fin algorithme
```

Dans les entrées de l'algorithme, les agents exportés représentent les agents ayant migré vers un autre processus grâce à l'algorithme 1. A ce stade de la distribution, ces agents et leurs voisinages sont encore complets au niveau du processus courant, contrairement à ceux des agents importés correspondant. L'objectif de cet algorithme consiste donc à compléter ces voisinages pour assurer la continuité des données.

Le voisinage d'un agent est toujours envoyé depuis le processus sur lequel cet agent est *local* (ligne 7), car c'est le seul processus sur lequel il est garanti que le voisinage de l'agent soit complètement représenté : en effet, il n'est pas nécessaire de maintenir le voisinage des agents *délégués*, qui ne sont pas exécutés. La requête du voisinage des agents exportés peut dépendre de la définition des agents, du modèle, de la représentation

de l'environnement ou encore des projections utilisées. Dans tous les cas, le voisinage exporté depuis le processus en cours n'est pas seulement constitué d'agents *locaux*. En effet, le voisinage d'un agent peut contenir des agents *délégués*, exécutés par un processus différent de celui sur lequel est exporté l'agent. C'est pourquoi il est également nécessaire d'exporter la *localisation* des agents *délégués* exportés (ligne 8), afin que le processus récepteur connaisse le processus sur lequel sont exécutés les agents *délégués* nouvellement créés (lignes 15 et 16). Afin d'éviter les duplications d'agents, un agent *délégué* n'est effectivement créé que s'il n'est pas déjà représenté par un agent *local* ou *délégué* au niveau du processus courant (ligne 14). Dans tous les cas, la représentation disponible est utilisée pour mettre à jour le voisinage de l'agent importé correspondant au voisinage reçu (ligne 18). Cette mise à jour prend des formes aussi diverses que la requête des voisinages, selon le contexte d'implémentation : créer le lien correspondant dans le graphe d'interactions, mettre à jour des attributs référençant l'agent, ou encore ne rien faire, par exemple si le voisinage n'est obtenu qu'à l'exécution par une requête spatiale sur tous les agents actuellement représentés sur le processus.

Nous introduisons enfin l'algorithme 3, qui permet de supprimer les agents *délégués* qui n'appartiennent plus à aucun voisinage d'agent *local*. Cette proposition n'est qu'un exemple basique : il est possible de concevoir des procédés plus complexes de mise en cache, afin d'éviter de recréer des agents *délégués* qui seraient à nouveau importés lors des prochaines migrations, ou d'appliquer l'algorithme seulement en cas de besoin et pas à chaque migration, à la manière des algorithmes de récupération de mémoire (« Garbage Collector »). Dans tous les cas, il est nécessaire d'implémenter une procédure de gestion de ces agents, sans quoi la quantité d'agents *délégués* créés au fil des migrations peut croitre jusqu'à représenter la totalité des agents de la simulation, au risque de saturer la mémoire.

Algorithme 3 Nettoyage des agents *délégués*.

Entrée:
1: Liste des agents *délégués*
2: Voisinages des agents *locaux*
Sortie:
3: Liste d'agents *délégués* réduite de ses éléments "inutiles"

```
4: algorithme ClearDelegates
5:     pour chaque agent des agents délégués faire
6:         si l'agent n'appartient à aucun des voisinages alors
7:             Supprimer l'agent délégué
8:         fin si
9:     fin pour
10: fin algorithme
```

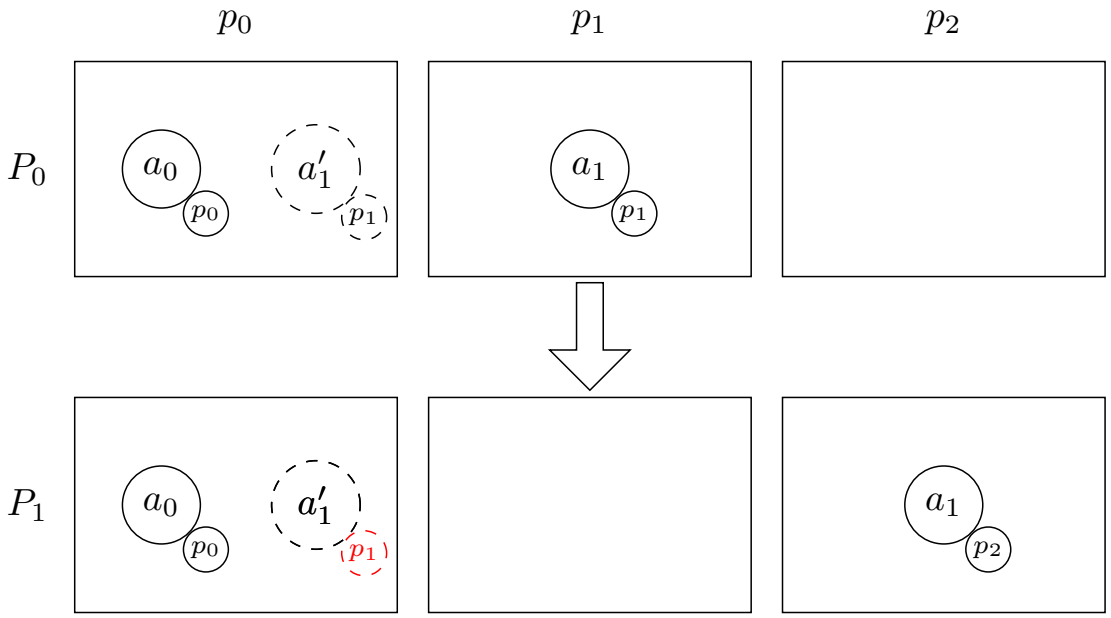


FIGURE 3.5. – Exemple de partitionnement sur 3 processus, et migration de l'agent a_1 vers le processus p_2.

3.3.3. Gestion de la localisation

Afin d'assurer la synchronisation des données, il est nécessaire de connaitre le processus sur lequel sont exécutés chacun des agents *distants* représentés par les agents *délégués*.

La connaissance de la localisation des agents *distants* peut se baser sur des informations contextuelles spécifiques à certains modèles. Par exemple, dans le cadre d'un modèle spatial et d'une décomposition à base de grille, où une zone géographique fixe est associée à chaque processus, on peut déduire la localisation d'un agent *délégué* à partir de ses coordonnées géographiques, sans qu'aucune communication additionnelle ne soit nécessaire. Bien qu'efficaces, ces solutions ne sont pas généralisables à tout modèle Multi-Agents, et dépendent du partitionnement du modèle (dans l'exemple précédent, un partitionnement spatial est nécessaire).

La localisation des agents *délégués* est initialisée lors de leur création, comme vu précédemment avec l'algorithme 2. Cependant, ce mécanisme ne suffit pas à mettre à jour la localisation de l'agent au niveau de tous les processus qui en détiennent une représentation. La figure 3.5 présente un exemple permettant d'illustrer ce problème, à partir d'un SMA sans structure particulière, partitionné sur trois processus, avec seulement deux agents. Nous considérons une partition P_1 dans laquelle l'agent a_0 a accès à l'agent *délégué* a'_1, qui représente a_1. Nous supposons que la localisation de l'agent *délégué* a'_1 est bien initialisée au processus p_1. La migration de l'agent a_1 vers le processus p_2 est alors effectuée pour mettre en place la partition P_2 : selon le processus de migration spécifié, qui initialise seulement la localisation des agents importés, la localisation de la représentation a'_1 n'est pas mise à jour et représente alors une information erronée.

L'algorithme 4, qui représente une implémentation du `LocationManager` de la figure 3.4, permet de mettre à jour la localisation des agents *délégués* pour tout type de modèle Multi-Agents, indépendamment du partitionnement. La solution proposée consiste tout d'abord à associer à chaque agent, lors de sa création, un *processus gestionnaire* fixe au cours de la simulation, dont le rôle consiste à maintenir à jour, à chaque distribution du système, la localisation des agents qu'il gère, même si ces agents ne sont pas nécessairement *locaux* au processus gestionnaire. Il est possible d'ajouter au cours de la simulation de nouveaux agents au gestionnaire, ou de les supprimer lorsqu'ils sont retirés de la simulation.

Dans nos expérimentations, le processus sur lequel est créé l'agent est utilisé comme gestionnaire. Cette méthode possède l'avantage de distribuer naturellement la gestion de la localisation des agents, contrairement au cas où un unique processus est utilisé en tant que gestionnaire par exemple. Des solutions plus complexes et potentiellement dynamiques pourraient être mises en place afin d'assurer un meilleur équilibrage de la gestion de la localisation des agents *délégués*, mais les expérimentations menées avec différents types de modèles n'ont pas montré un coût en performance excessif associé à cette méthode.

Suite à l'importation des agents, la mise à jour de la localisation des agents *délégués* consiste en trois étapes.

1. Envoyer aux processus gestionnaires la nouvelle localisation des agents importés (ligne 6). Le processus gestionnaire met alors à jour la localisation actuel des agents qu'il gère (ligne 10).
2. Demander aux processus gestionnaires la localisation de tous les agents *délégués* (ligne 14). Cette étape suppose que le processus gestionnaire de chaque agent est connu par tous les processus pour tout agent à tout moment de la simulation. C'est pourquoi nous utilisons en tant que gestionnaire le processus d'origine de l'agent, contenu dans son identifiant. D'autres solutions sont facilement envisageables.
3. Les processus gestionnaires peuvent ensuite répondre aux requêtes (ligne 18), et le processus en cours peut enfin mettre à jour la localisation de ses agents *délégués* (ligne 22).

L'inconvénient de cet algorithme est la réalisation systématique d'une requête pour chaque agent délégué, même si leur localisation n'a pas changé. Le recours à un nombre constant de communications collectives [2] entre les processus constitue cependant un avantage qui permet généralement de bonnes performances et favorise le passage à l'échelle quand le nombre de processus augmente. Le coût observé de l'algorithme dans nos expérimentations s'est systématiquement avéré négligeable. Dans tous les cas, cette solution n'est qu'une proposition. Il est possible d'en imaginer d'autres, par exemple basées sur la recherche à la volée de la localisation des agents.

2. L'envoi et la réception des requêtes peut par exemple être réalisé en une seule communication de type `MPI_AllToAll`

Algorithme 4 Gestion de la localisation des agents *délégués*.

Entrée:

Liste d'agents *délégués*
Liste d'agents *locaux* importés

Sortie:

Localisation à jour des agents *délégués*

algorithme UPDATELOCATIONS
    pour chaque agent importé **faire**
        ENVOYER À gestionnaire de l'agent : (id agent, processus courant)
    fin pour
    RECEVOIR DEPUIS tous les processus : localisations à jour
    pour chaque (id agent géré, processus) des localisations à jour **faire**
        Mettre à jour la localisation associée à l'agent géré
    fin pour

    pour chaque agent délégué **faire**
        ENVOYER À gestionnaire de l'agent : (id agent, processus courant)
    fin pour
    RECEVOIR DEPUIS tous les processus : requêtes
    pour chaque (id agent, processus) des requêtes **faire**
        ENVOYER À processus : (id agent, localisation à jour de l'agent)
    fin pour
    RECEVOIR DEPUIS tous les processus : localisations à jour
    pour chaque (id agent, processus) des localisations à jour **faire**
        Mettre à jour la localisation de l'agent délégué
    fin pour
fin algorithme

3.3.4. Distribution

Nous définissons enfin l'algorithme 5 de distribution d'un SMA, grâce à des appels aux algorithmes 1, 2, 3 et 4 précédemment définis. Quelles que soient les partitions initiales et finales du modèle, la continuité des données est préservée grâce au mécanisme de création des agents *délégués*. Le procédé de distribution est donc un atout dans l'abstraction de l'architecture distribuée pour l'utilisateur final, qui pourra spécifier les comportements des agents indépendamment de la distribution, gérée en interne par la plateforme de simulation. D'autre part, les algorithmes décrits ne sont que des propositions qui peuvent facilement faire l'objet d'alternatives et d'optimisations. Néanmoins, les différentes étapes de la distribution semblent nécessaires à toute distribution de SMA devant supporter la continuité des données quel que soit le partitionnement.

Algorithme 5 Distribution d'un modèle Multi-Agents.

Entrée:
1: Modèle distribué
2: Nouvelle Partition
Sortie:
3: Modèle distribué selon la nouvelle partition

4: **algorithme** DISTRIBUTION
5: MIGRATION(modèle, nouvelle partition)
6: CREATEDELEGATES(agents exportés, agents importés)
7: CLEARDELEGATES(agents délégués)
8: UPDATELOCATIONS(agents délégués, agents importés)
9: **fin algorithme**

Le procédé de création de nouvelles partitions pourra éventuellement être spécifié par l'utilisateur, selon des critères qui lui sont propres. Cependant, dans notre démarche d'abstraction de la distribution, l'objectif est plutôt d'automatiser en interne la création des nouvelles partitions grâce à des algorithmes d'équilibrage de charge, décrits dans le chapitre 4.

3.4. Cas de la représentation à base de graphe

Tous les modèles Multi-Agents ont comme caractéristique commune de représenter des agents interagissant entre eux. Dans le cadre de notre hypothèse où l'environnement est lui-même représenté par un ensemble d'agents, les interactions avec l'environnement sont assimilées à des interactions entre agents. La structure de graphe ne nécessitant pas d'information contextuelle supplémentaire (comme le type des agents, la taille de l'environnement, la position des agents, etc), elle est capable de représenter tous les SMA.

3.4.1. Principe

Pour construire une représentation à base de graphe d'un SMA arbitraire, il suffit d'associer un nœud à chaque agent, et de construire les liens nécessaires en considérant qu'un agent ne peut interagir qu'avec ceux auxquels il est connecté.

La construction du graphe est triviale dans le cas d'un modèle à base de graphe, mais peut aussi s'étendre aux modèles spatiaux. Dans le cadre d'un modèle à base de grille, l'environnement peut par exemple être représenté par des cellules connectées à leurs voisines, et la localisation des agents peut être représentée par un lien vers la cellule dans laquelle il se trouve. Des liens supplémentaires sont alors créés entre l'agent et ceux localisés dans les cellules voisines, en fonction de son champs de perception, pour lui permettre d'interagir avec eux. Cette méthode est extensible au cas où les agents ne se déplacent pas sur une grille mais sur un graphe arbitraire.

3.4.2. Motivations

La représentation à base de graphe peut garantir la distribution et la continuité de tous les SMA, y compris les modèles spatiaux, et ce indépendamment du partitionnement. En effet, n'importe quel nœud peut être associé à n'importe quel processus sans compromettre l'intégrité de son voisinage, ce qui constitue un gain en flexibilité notable par rapport aux distributions déjà évoquées des modèles à base de grille qui imposent d'associer à chaque processus une portion fixée et continue de l'environnement. Avec la structure de graphe, il est par exemple possible d'envisager des distributions où les agents localisés dans une même cellule ne sont pas exécutés sur le même processus, ce qui n'est pas possible avec une distribution à base de grille.

Comme l'ont montré nos expérimentations, la structure de graphe n'est clairement pas optimale pour représenter tous les modèles, notamment les modèles spatiaux à base de grille, qui nécessitent la construction de nombreux liens de manière dynamique. Cependant, la capacité de cette structure à représenter tous les SMA nous permet d'assurer la distribution de tous les modèles grâce à l'implémentation des algorithmes de distribution pour la seule structure de graphe.

Ainsi le choix a été fait, dans le cadre du développement de la plateforme FPMAS, d'utiliser une structure interne de graphe distribué pour représenter les SMA [29]. Afin de faciliter la représentation des interactions potentielles, le graphe est orienté. Chaque agent a alors accès à tous les liens sortants et entrants du nœud qui le représente et aux agents correspondants, mais l'accès à tout autre lien ou agent n'est pas garanti. Il est possible d'associer à chaque lien un poids et un type d'interaction, par exemple avec un entier ou une énumération. Il peut donc exister plusieurs liens avec des types différents d'un agent vers un autre. Ce choix n'a pas vocation à motiver une quelconque technique de modélisation : il est par exemple possible de spécifier un environnement de type grille et les champs de perception des agents indépendamment du concept de graphe, c'est alors à la plateforme de traduire ces fonctionnalités en termes de construction et de mise à jour du graphe interne.

3.4.3. Spécialisation des algorithmes

Les algorithmes précédents sont volontairement spécifiés de la manière la plus abstraite possible, mais leur implémentation nécessite une adaptation au contexte de simulation. Nous présentons ici un exemple de spécialisation des algorithmes de distribution dans le cas d'une représentation à base de graphe des SMA.

Migration

Le processus de migration décrit dans l'algorithme 1 est applicable en l'état à la représentation à base de graphe, en considérant la migration des agents *locaux* comme la migration des nœuds qui les représentent.

Création et nettoyage des agents délégués

Pour implémenter l'algorithme de création des agents *délégués* dans le cas d'un SMA représenté par un graphe, il suffit de considérer le voisinage de l'agent comme l'ensemble des agents qui lui sont connectés par des liens entrants ou sortants. Afin d'ajouter les agents importés aux voisinages, il est donc nécessaire de migrer les liens associés. L'algorithme 6 représente ainsi un exemple de spécialisation de l'algorithme 2 de création des agents *délégués* dans le cas d'une représentation à base de graphe.

Dans le cas d'un graphe, l'algorithme de nettoyage des agents *délégués* inutiles (algorithme 3) consiste simplement à supprimer tous les nœuds *délégués* dont aucun voisin dans le graphe n'est *local*, ainsi que les liens associés.

Gestion de la localisation

L'algorithme 4 de gestion de la localisation est nativement applicable au graphe distribué, en assimilant les agents aux nœuds qui les représentent.

3.4.4. Exemple de distribution

La figure 3.6 présente un exemple de graphe représentant un SMA. La figure 3.7 présente un exemple de distribution initiale de ce graphe sur trois processus, en mettant en œuvre le concept de continuité des données. Il n'est pas nécessaire d'ajouter les liens entre nœuds *délégués* aux graphes, car seuls les agents représentés par des nœuds *locaux* sont exécutés et peuvent être amenés à interagir avec leurs voisins, *locaux* ou *délégués*. Le partitionnement est arbitraire, et ne fait l'objet d'aucune contrainte. Il est par exemple possible d'assigner librement le nœuds a_4 au processus 0, même s'il n'est pas connecté à un autre nœud *local*.

Une attention particulière est ici apportée à la transparence de la distribution. On constate par exemple que le voisinage de a_0 est constitué de a_5, a_1 et a_2, et ce quel que soit la distribution du système. Dans ce cas, on observe que a_1 et a_2 sont *délégués*, alors que a_5 est *local*, mais la constitution du voisinage de a_0 est parfaitement préservée : du point de vue de l'utilisateur, a_0 peut donc être amené à interagir avec chacun d'eux,

Algorithme 6 Création des agents *délégués* pour une représentation à base de graphe.

Entrée:

Agents exportés avec leur voisinage à jour
Agents importés sans voisinage

Sortie:

Agents importés avec leur voisinage à jour

```
algorithme ImportAgent(graphe, agent, localisation de l'agent)
    si agent ∉ graphe alors
        Ajouter l'agent au graphe dans un nœud délégué
        Initialiser la localisation de l'agent
    fin si
fin algorithme

algorithme CreateDelegates
    pour chaque agent des agents exportés faire
        p ← nouvelle localisation de l'agent
        pour chaque lien entrant de l'agent faire
            si localisation agent cible ≠ p alors
                envoyer à p : (lien, agent cible, localisation cible)
            fin si
        fin pour
        pour chaque lien sortant de l'agent faire
            si localisation agent source ≠ p alors
                envoyer à p : (lien, agent source, localisation source)
            fin si
        fin pour
    fin pour
    recevoir depuis tous les processus : voisinages des agents importés
    pour chaque lien sortant reçu faire
        ImportAgent(graphe, agent cible, localisation agent cible)
        Insérer le lien dans le graphe
    fin pour
    pour chaque lien entrant reçu faire
        ImportAgent(graphe, agent source, localisation agent source)
        Insérer le lien dans le graphe
    fin pour
fin algorithme
```

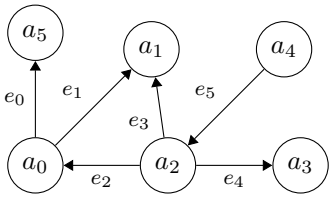
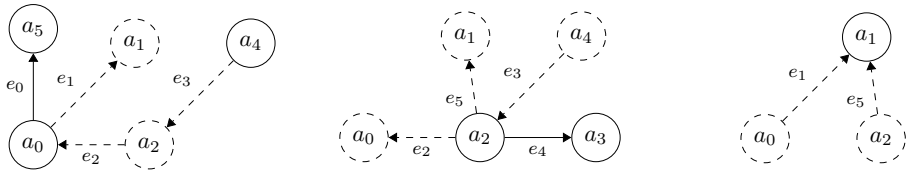


FIGURE 3.6. – Graphe d'exemple représentant un SMA. Un lien de a_x vers a_y indique que a_x peut interagir avec a_y.



(a) Processus 0 (b) Processus 1 (c) Processus 2

FIGURE 3.7. – Exemple de distribution du graphe d'exemple sur 3 processus. Les nœuds pleins sont *locaux*, ceux en pointillés sont *délégués*.

sans se soucier de l'état *local* ou *délégué* de chaque voisin. Les spécificités liées aux nœuds *délégués*, comme la nécessité de mettre en place des communications avec les autres processus, doivent être gérées en interne par la plateforme. Ce problème est traité indépendamment de la structure de graphe dans le chapitre 5.

Nous illustrons ensuite le déroulement de l'algorithme de distribution 5 dans le cas d'une représentation à base de graphe du modèle présenté à la figure 3.6.

L'objectif consiste à mettre en place une nouvelle partition du modèle distribué initial de la figure 3.7, ayant pour effet la migration de l'agent a_0 vers le processus 2. Pour rappel, il suffit d'avoir accès à l'échelle de chaque processus à la portion de la partition qui concerne ses nœuds *locaux*. La localisation des nœuds non spécifiés est supposée inchangée. Les nouvelles partitions suivantes peuvent donc être utilisées en entrées de l'algorithme de distribution sur chaque processus :

— Processus 0 : $\{a_0 : 2\}$
— Processus 1 : $\{\}$
— Processus 2 : $\{\}$

La figure 3.8 illustre la migration du nœud a_0 grâce à l'algorithme de migration 1. Le nœud *local* a_0 est remplacé par un nœud *délégué* au niveau du processus 0, et le nœud *délégué* a_0 est remplacé par un nœud *local* sur le processus 2.

Il est ensuite nécessaire d'appliquer l'algorithme de création des agents *délégués* 6 pour migrer les liens et construire le voisinage de a_0 sur le processus 2, comme présenté sur la figure 3.9. Seuls les liens e_0 et e_2 sont exportés : l'agent *délégué* a_1 permet de vérifier que a_1 est localisé sur le processus 2, le lien e_1 n'a donc pas besoin d'être exporté car il est nécessairement déjà représenté sur le processus 2. L'importation du lien e_2 se fait par insertion d'un lien entre a_0 et l'agent *délégué* existant a_2. Pour le lien e_1, un agent

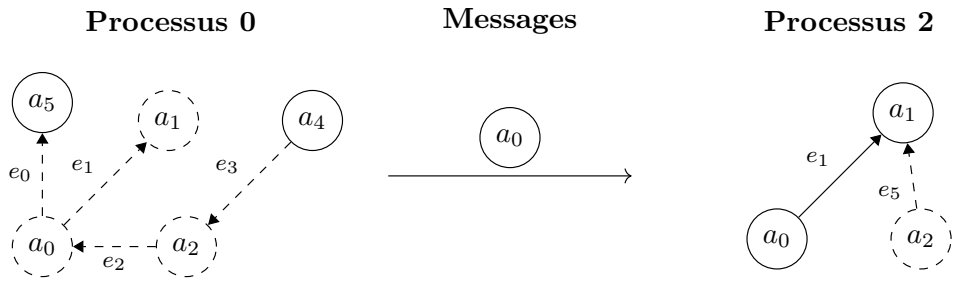


FIGURE 3.8. – Migration du nœud a_0 du processus 0 vers le processus 2

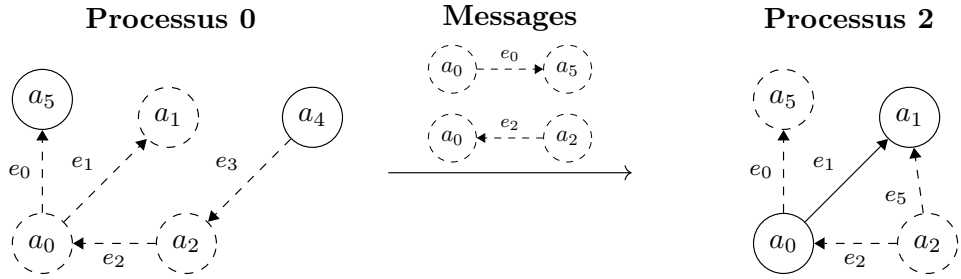


FIGURE 3.9. – Envoi des liens nécessaires et création de noeuds *délégués* suite à la migration de a_0.

délégué a_5 est d'abord créé avant d'insérer le lien, assurant la continuité des données et l'intégrité du voisinage de a_0, nouvellement importé.

Pour finir, la figure 3.10 montre l'application de l'algorithme de nettoyage 3 au processus 0. L'agent *délégué* a_1, inaccessible par tous les nœuds *locaux*, est supprimé. Le processus 2 n'est pas représenté car aucun nœud ne peut y être supprimé.

Le processus 1 n'a pas été représenté sur les figures, car aucun changement du graphe n'est nécessaire. Cependant, la figure 3.6 montre que le processus 1 possède une représentation de l'agent a_0 : il est donc nécessaire de mettre à jour la localisation de a_0, comme évoqué à la section 3.3.3. L'algorithme de gestion de la localisation 4 étant complètement indépendant de la structure de graphe, son application n'est pas représentée ici.

3.5. Autres problèmes de distribution

Nous abordons ici brièvement d'autres problématiques liées à la simulation distribuée de manière générale, sans pour autant impacter la mise en place de la distribution d'une simulation de SMA.

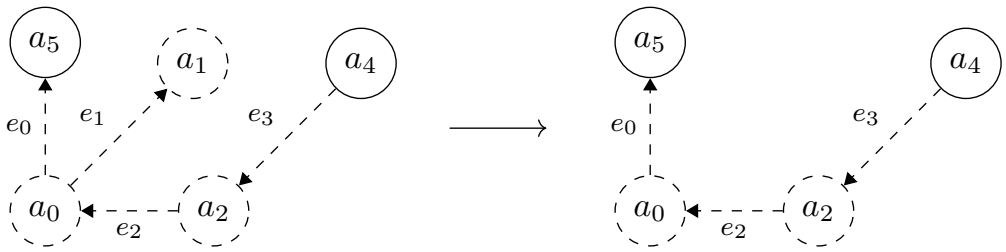


FIGURE 3.10. – Nettoyage des agents *délégués* sur le processus 0.

3.5.1. Sérialisation des données

À de nombreuses reprises, les algorithmes précédents requièrent l'envoi et la réception d'agents ou d'autres données entre les processus. Nous pouvons légitimement abstraire ce procédé dans la spécification des algorithmes, en supposant qu'il est toujours possible de transmettre les données requises. Compte tenu du contexte distribué, il est cependant toujours nécessaire de mettre en place une solution concrète pour résoudre ce problème. Ainsi nous introduisons ici quelques éléments de réflexion issus de notre expérience personnelle ou des outils existants.

La *sérialisation* d'un objet ou d'un agent consiste à en construire une représentation transmissible via un réseau (par exemple une suite d'octets ou de caractères), à partir de laquelle il est possible de reconstruire l'objet d'origine grâce au processus de *désérialisation*.

Dans un contexte de simulation distribuée, la sérialisation représente plus qu'une contrainte technique. En effet, les processus ne pouvant communiquer entre eux que par messages via un réseau, il est par définition nécessaire de mettre en place une méthode de sérialisation des agents pour permettre les migrations et interactions entre les processus.

De nombreuses techniques de sérialisation existent. Les formats JSON ou XML sont des exemples de méthodes génériques largement utilisées. La plateforme Repast HPC utilise quant à elle la librairie de sérialisation d'objet C++ *Boost Serialization Library* [9]. Le protocole MPI lui-même propose également une interface de sérialisation bas niveau. D'autres méthodes sont développées dans des contextes plus spécifiques. Nous ne cherchons pas ici à appuyer le choix d'une méthode ou d'une autre, mais d'attirer l'attention du développeur de simulation distribuée de SMA sur le fait qu'une technique de sérialisation des agents doit nécessairement être mise en place.

Dans le contexte du développement de FPMAS, nous avons d'abord basé le processus de sérialisation sur le format JSON et la librairie *Json for Modern C++* [7]. De manière générale, l'intérêt du format JSON réside dans sa lisibilité par l'humain et son interopérabilité. Il est par exemple possible d'utiliser les données sérialisées de manière interne à la plateforme d'une part pour la transmission de messages entre les processus, et de manière externe d'autre part, en entrée d'outils qui supportent le format JSON, par exemple pour effectuer de l'affichage web. La librairie utilisée propose en

outre une interface de sérialisation haut niveau, ce qui est particulièrement compatible avec nos objectifs d'accessibilité et de transparence de la distribution. Cependant, nos expérimentations ont montré un coût très important de l'utilisation de cette librairie, à la fois en mémoire et en temps de calcul. En effet il a été observé que le processus de sérialisation requiert l'allocation en mémoire d'un objet `json` pour chaque champs de données, ce qui s'avère extrêmement coûteux dans un contexte de simulation large échelle, où la sérialisation de dizaines de milliers d'agents est nécessaire à chaque pas de temps.

Pour pallier ce problème, nous avons conçu une technique de sérialisation nommée *ObjectPack* [4], dont l'architecture est très largement inspirée de la librairie précédente, afin d'effectuer la sérialisation des données dans un format binaire optimisé pour l'envoi de message grâce au protocole MPI, tout en proposant une interface haut niveau. L'optimisation et la maitrise exacte des coûts en mémoire et en temps de calcul du processus de sérialisation, qui s'avère critique, ont constitué une motivation essentielle pour développer notre propre méthode de sérialisation. Ainsi la librairie *ObjectPack* est actuellement interne à FPMAS, mais pourrait être rendue indépendante à l'avenir. Le principe de base consiste à faire une copie mémoire directe, octet par octet, des types fondamentaux (entiers, réels, caractères...) dans un espace mémoire à transmettre grâce au protocole MPI. Pour chaque type de données, nous définissons trois méthodes pour :

1. Requérir l'espace mémoire nécessaire pour écrire les données ;
2. Écrire les données dans un espace mémoire dédié ;
3. Lire les données depuis un espace mémoire donné.

Nous supposons que chaque type de données peut lui-même être constitué d'autres types. La première passe de l'algorithme de sérialisation consiste à récursivement requérir la taille nécessaire pour écrire chaque type jusqu'à atteindre les types fondamentaux. Un espace mémoire pouvant contenir l'intégralité des données est alors réservé en une seule allocation, ce qui permet une meilleure efficacité en mémoire et en temps de calcul. La seconde passe de l'algorithme consiste à écrire chaque pièce de données selon le schéma spécifié par l'utilisateur dans l'espace mémoire qui lui a été alloué. Le procédé de désérialisation est similaire, les copies mémoires directes étant réalisées de l'espace mémoire vers des instances de types fondamentaux. Le procédé de sérialisation détaillé et des exemples sont disponibles à l'annexe A.

Les méthodes de sérialisation sont déclarées de manière statique et réalisent directement les lectures et écritures dans l'espace mémoire alloué. Le processus de sérialisation nécessite donc l'allocation d'un nombre constant et limité d'objets, indépendamment des structures à sérialiser, contrairement à la librairie *Json for Modern C++*.

Quelle que soit la méthode de sérialisation utilisée, la spécification des règles de sérialisation des agents reste indépendante de la distribution du point de vue de l'utilisateur. En effet, ces règles ne dépendent pas du contexte distribué, mais peuvent être utilisées en interne et de manière transparente par la plateforme de simulation.

3.5.2. Génération de nombres aléatoires

La génération de nombres aléatoires est un enjeu important de la simulation numérique, en séquentiel comme en distribué, particulièrement dans le cas des SMA où le comportement des agents se base souvent sur des phénomènes aléatoires.

Il existe deux types de générateurs de nombres aléatoires. Les générateurs physiques se basent sur des mesures de bruits divers pour fournir des séquences de nombres aléatoires non reproductibles. En comparaison, les générateurs pseudo-aléatoires produisent grâce à des algorithmes des séquences en apparence aléatoires, mais parfaitement reproductibles et prévisibles connaissant l'état actuel du générateur. Des exemples classiques d'algorithmes de génération de nombre pseudo-aléatoires incluent le générateur congruentiel linéaire [99] et le Mersenne Twister [89]. Nous considérons ici des générateurs de nombres pseudo-aléatoires pouvant être configurés grâce à une graine qui permet de modifier la séquence de nombres générés. Le choix du type de générateur dépend des contraintes de l'utilisateur ou de l'environnement de simulation. Dans le cadre de la conception d'une plateforme distribuée générique, il est pertinent de se questionner sur la mise en place d'une génération de nombres aléatoires distribuée.

La distribution des générateurs physiques est triviale et ne laisse pas le choix : des séquences aléatoires différentes sur chaque processus sont toujours générées à chaque exécution. Les générateurs pseudo-aléatoires peuvent faire l'objet de deux schémas de distribution. Dans le premier cas, les mêmes séquences sont générées sur chaque processus. Il suffit pour cela d'utiliser une instance de générateur déterministe sur chaque processus, tous initialisés avec la même graine. Ce mécanisme peut s'avérer intéressant pour construire des ensembles aléatoires identiques sur tous les processus, indépendamment du nombre de processus, comme discuté au chapitre 5. Ce schéma peut cependant induire un biais significatif lorsque les processus doivent utiliser des séquences de nombres indépendantes, par exemple pour le déplacement aléatoire des agents qu'ils exécutent. C'est pourquoi dans le second schéma des séquences de nombres aléatoires différentes mais déterministes sont générées sur chaque processus. Une solution consiste à utiliser un générateur pseudo-aléatoire sur chaque processus, comme présenté dans l'algorithme 7. Chaque générateur est initialisé à partir d'une graine elle-même générée aléatoirement et de manière déterministe à partir d'une graine spécifiée par l'utilisateur, ce qui permet d'obtenir des séquences de nombres indépendantes sur chaque processus mais déterminées par la graine initiale. Pour des raisons de simplicité, l'algorithme présenté ici initialise tout le tableau de graines sur tous les processus. Dans la pratique, il suffit de générer seulement les p premières graines en ne conservant que celle nécessaire au processus courant.

Du point de vue de l'utilisateur, la distribution est transparente : il lui suffit d'initialiser un générateur distribué à partir d'une seule graine pour obtenir une génération de nombres aléatoires cohérente sur tous les processus, reproductible tant que le nombre de processus est fixé. L'algorithme est applicable à tout générateur pseudo-aléatoire séquentiel.

Algorithme 7 Génération de nombres aléatoires déterministes et distribuées

Entrée:

1: générateur de nombre pseudo-aléatoires
2: $|P|$, nombre de processus
3: p, indice du processus courant

4: **algorithme** INITIALISER(graine aléatoire)
5: générateur de graines ← générateur initialisé avec la graine aléatoire
6: graines ← tableau de $|P|$ graines aléatoires
7: **pour** i de 0 à $n-1$ **faire**
8: graines[i] ← graine générée par le générateur de graines
9: **fin pour**
10: générateur local ← générateur initialisé avec graines[p]
11: **fin algorithme**

12: **algorithme** GÉNÉRER
13: **retourner** valeur générée par le générateur local
14: **fin algorithme**

3.6. Synthèse

Toute mise en place de simulation distribuée de SMA générique fait face à certaines difficultés. Les problèmes de migration, de création d'agents *délégués*, de maintien à jour des localisations et de nettoyage des agents *délégués* ont notamment été identifiés pour permettre la mise en place de la distribution des SMA. Les questions de la sérialisation des données et de la génération de nombres aléatoires mettent en avant d'autres problèmes de simulation distribuée génériques. Des solutions dont le niveau d'abstraction permet leur applicabilité à tout SMA ont été proposées, même si la prise en compte de certains contextes spécifiques, comme les modèles spatiaux, peuvent donner lieu à des variantes optimisées. Un exemple de mise en place des algorithmes de distribution dans le cadre d'une structure de données à base de graphe a notamment été présentée. L'utilisation d'un graphe permet de faciliter l'implémentation de la distribution des modèles, notamment grâce à l'accès trivial du voisinage des agents dans une structure de graphe. La mise en place de la distribution d'un modèle ne suffit cependant pas à réaliser l'exécution distribuée d'une simulation de SMA. La résolution d'autres problèmes est abordée dans la suite, en commençant par l'équilibrage de charge.

Chapitre 4.

Équilibrage de charge

Dans la partie précédente, nous avons présenté une revue non exhaustive de travaux théoriques et pratiques autour du problème de l'équilibrage de charge, applicables plus ou moins directement à la simulation distribuée de SMA. Afin de les unifier, nous définissons d'abord dans la section 4.1 le problème théorique de l'équilibrage de charge adapté à la simulation distribuée de SMA. Nous proposons ensuite dans la section 4.2 une architecture logicielle générique et extensible sous forme d'interface permettant l'implémentation de divers algorithmes d'équilibrage de charge. Les différents types de modèles et d'environnements considérés ainsi qu'un *Méta-Modèle* permettant une configuration expérimentale générique et flexible sont introduits dans la section 4.3. La définition de plusieurs algorithmes d'équilibrage de charge et leur analyse qualitative sont présentées dans la section 4.4. Les expérimentations de la section 4.5 présentent enfin des estimations de performances des algorithmes d'équilibrage implémentés dans FPMAS pour différents types de modèles.

L'objectif de nos travaux ne consiste pas à concevoir l'algorithme d'équilibrage de charge optimal pour chaque modèle, mais plutôt de démontrer l'utilité pour une plateforme de simulation de SMA générique de proposer divers algorithmes d'équilibrage pour s'adapter aux modèles des utilisateurs. Nous montrons également comment l'architecture logicielle proposée permet d'implémenter de manière générique et flexible divers algorithmes d'équilibrage de charge, notamment au travers d'exemples d'application de la librairie de partitionnement de graphe Zoltan à l'équilibrage de simulations distribuées de SMA.

Table des matières

4.1. Approche théorique de l'équilibrage de charge

L'*équilibrage de charge* a déjà été défini au chapitre 2 comme un algorithme qui consiste à assigner chaque agent à un processus au cours de la simulation en faisant appel de manière ponctuelle (équilibrage *statique*) ou itérative (équilibrage *dynamique*) à une méthode de *partitionnement*, chargée de construire des *partitions*.

Nous nuançons ici la définition de ce partitionnement selon deux cas d'application. Dans le cas du *partitionnement*, l'algorithme cherche à optimiser l'équilibrage de la simulation indépendamment de la distribution actuelle du modèle. Dans le cas du *repartitionnement*, l'algorithme prend en compte la distribution actuelle du modèle afin notamment de limiter la migration des données entre les processus, qui a elle-même un coût.

Ainsi l'algorithme d'équilibrage de charge peut, au cours de la simulation, faire appel à la méthode de création de partitions selon deux modes, *partitionnement* ou *repartitionnement*, que l'équilibrage soit *statique* ou *dynamique*. Par exemple, l'application d'une méthode de *partitionnement* par un équilibrage *dynamique* consiste à équilibrer la simulation de manière récurrente sans prendre en compte le partitionnement actuel de la simulation. Cette méthode peut s'avérer pertinente lorsque le déséquilibre devient trop marqué pour être résolu par les modifications locales des méthodes de *repartitionnement*. L'application d'une méthode de *repartitionnement* par un équilibrage *statique* consiste quant à elle à équilibrer la simulation en prenant en compte la partition initiale du modèle. Le comportement par défaut des algorithmes d'équilibrage de charge de la plateforme FPMAS consiste à appliquer une méthode de *partitionnement* à la première itération, puis à appliquer une méthode de *repartitionnement* aux itérations suivantes si une application récurrente est demandée par l'utilisateur, pour obtenir un équilibrage *dynamique*. Certains algorithmes, comme le partitionnement aléatoire, ne font pas de réelles distinctions entre les deux modes. Les travaux de Wang et al. [127] décrivent un équilibrage dynamique des simulations distribuées de SMA avec une distinction claire entre les méthodes de partitionnement et de repartitionnement. Notre approche reste compatible avec ces deux exemples.

4.1.1. Problème de partitionnement

Pour définir le problème du *partitionnement*, chaque agent a_i est associé à un poids w_i, censé représenter le temps d'exécution relatif de chaque agent par rapport aux autres. Nous définissons alors les éléments suivants, largement inspirés des travaux de Lui et Chan [78] dans le domaine des environnements virtuels distribués et adaptés à notre contexte :

- P : ensemble de processus ;
- N_P : nombre de processus ;
- $\mathcal{S} : A \to P$: partition de l'ensemble des agents ;
- $u_{\mathcal{S},p} = \{a_i, \mathcal{S}(a_i) = p\}$ l'ensemble des agents associés au processus $p \in P$ selon la partition $\mathcal{S}$;

— $W_{\mathcal{S},p} = \sum_{a_i \in u_{\mathcal{S},p}} w_i$: charge de calcul totale associée au processus $p \in P$;

— $W^*_{\mathcal{S}} = \frac{\sum_{p \in P} W_{\mathcal{S},p}}{N_P} = \frac{\sum_{a_i \in A} w_i}{N_P}$: charge de calcul moyenne associée à chaque processus.

Dès lors, nous définissons deux types de coûts associés à une partition : le coût en charge de calcul et le coût en communications.

Le coût en charge de calcul de la partition $\mathcal{S}$ est défini par l'équation 4.1.

$$C^W_{\mathcal{S}} = \sum_{p \in P} |W_{\mathcal{S},p} - W^*_{\mathcal{S}}| \tag{4.1}$$

Cette définition n'est pas unique. Il est par exemple possible de définir un autre coût par le maximum des écarts à la charge de calcul moyenne. Dans les deux cas, minimiser le coût revient à minimiser l'écart entre la charge associée à chaque processus et la charge moyenne, de sorte à égaliser la charge de calcul associée à tous les processus.

Nous définissons ensuite :

— $\gamma_{\mathcal{S}} : A \times A \to \mathbb{R}^+$ telle que $\gamma_{\mathcal{S}}(a_i, a_j)$ désigne le coût en temps des communications depuis l'agent a_i vers a_j dans une simulation distribuée selon la partition $\mathcal{S}$. Les communications n'étant pas symétriques, $\gamma_{\mathcal{S}}(a_i, a_j) \neq \gamma_{\mathcal{S}}(a_j, a_i)$.

Plusieurs cas sont notables.

— Si a_i et a_j sont associés au même processus, alors $\gamma_{\mathcal{S}}(a_i, a_j) = 0$.

— Sinon si a_j n'appartient pas au voisinage de a_i, c'est-à-dire que a_i n'est pas amené à interagir avec a_j d'après les règles du modèle, alors $\gamma_{\mathcal{S}}(a_i, a_j) = 0$.

— Sinon, a_i va pouvoir communiquer avec a_j par l'intermédiaire d'un agent *délégué* représentant a_j, conformément au principe de distribution décrit au chapitre précédent. Dès lors, $\gamma_{\mathcal{S}}(a_i, a_j) \geq 0$.

Le coût en communications de la partition $\mathcal{S}$ peut se définir ainsi :

$$C^L_{\mathcal{S}} = \sum_{a_i \in A} \sum_{a_j \in A} \gamma_{\mathcal{S}}(a_i, a_j) \tag{4.2}$$

Minimiser ce coût revient à limiter les communications entre les processus. Nous pouvons alors nous baser sur la formule générale de Lui et Chan [78] déjà évoquée au chapitre 4, de sorte que le problème du partitionnement consiste à trouver le partitionnement $\mathcal{S}$ minimisant la quantité suivante :

$$C_{\mathcal{S}} = C^W_{\mathcal{S}} + C^L_{\mathcal{S}} \tag{4.3}$$

Les paramètres W_1 et W_2 utilisés par Lui et Chan pour ajuster l'importance relative du coût en charge et en communication ne sont pas utilisés ici, car considérer seulement la minimisation de la somme globale suffit à notre propos.

4.1.2. Problème de repartitionnement

Le problème du partitionnement ne prend pas en compte la partition actuelle du modèle. D'où l'introduction de la notion de *repartitionnement*, étudiée notamment par

les concepteurs de Zoltan, une librairie de repartitionnement et d'équilibrage de charge dynamique d'hypergraphes déjà évoquée [34, 35, 6].

L'objectif du repartitionnement consiste à rétablir l'équilibre de la partition actuelle grâce à des modifications locales de sorte à limiter les migrations d'agents. Le repartitionnement prend donc en compte le coût de migration des agents, contrairement au partitionnement, qui risque d'entrainer des migrations inutiles et conséquentes même si l'équilibre de la charge est préservé. Par exemple, effectuer seulement une permutation des processus à partir d'une partition déjà équilibrée engendrerait la migration inutile de la totalité des agents, tout en préservant l'équilibre. Le repartitionnement permet d'une part de résoudre ce problème, et d'autre part d'éviter la migration de certains agents si leur coût de migration est supérieur au gain en performance estimé suite à leur migration.

Nous enrichissons les définitions précédentes avec des notions de repartitionnement introduites par Catalyurek et al. [35] pour les adapter à la simulation distribuée de SMA.

Soit une partition initiale $\mathcal{S}'$, et une partition objectif $\mathcal{S}$. Nous définissons :

— $\lambda_{\mathcal{S}',\mathcal{S}}(a_i)$: coût de migration de l'agent a_i dans le cadre de la distribution du modèle selon $\mathcal{S}$ en partant de la partition $\mathcal{S}'$;

— $C^M_{\mathcal{S}',\mathcal{S}} = \sum_{a_i \in A} \lambda_{\mathcal{S}',\mathcal{S}}(a_i)$: coût total de migration de $\mathcal{S}'$ à $\mathcal{S}$: ;

— α : facteur multiplicateur permettant d'ajuster l'importance de l'équilibrage de la partition par rapport aux coûts de migration. Ce paramètre permet notamment d'ajuster la comparaison entre le coût de migration d'un agent et le gain en performance induit par sa migration.

Le problème du repartitionnement, considérant une partition $\mathcal{S}'$ initiale, consiste alors à construire une partition $\mathcal{S}$ minimisant le coût suivant :

$$\begin{aligned} C_{\mathcal{S}',\mathcal{S}} &= \alpha(C^W_{\mathcal{S}} + C^L_{\mathcal{S}}) + C^M_{\mathcal{S}',\mathcal{S}} \\ &= \alpha C_{\mathcal{S}} + C^M_{\mathcal{S}',\mathcal{S}} \end{aligned} \tag{4.4}$$

4.2. Interface générique

Comme vu au chapitre 2, il existe une grande diversité d'algorithmes d'équilibrage de charge applicables au cas de la simulation distribuée de SMA. Compte tenu des définitions formelles précédentes des problèmes de partitionnement et repartitionnement, qui correspondent à des problèmes d'optimisation, les algorithmes d'optimisation génériques (algorithmes génétiques, Tabu search, optimisation par colonies de fourmis...) sont également applicables au problème de l'équilibrage de charge de SMA [125].

Tous les algorithmes ne sont pas applicables à tous les modèles, et tous n'exposent pas les mêmes performances selon le modèle simulé. Dans le cadre d'une plateforme de simulation distribuée de SMA qui se veut générique et efficace, le support pour différents algorithmes d'équilibrage de charge, voire la possibilité d'implémenter des algorithmes spécifiques à un modèle, semble donc nécessaire.

De plus, le meilleur algorithme d'équilibrage de charge n'est pas nécessairement celui produisant la partition de meilleur qualité, c'est-à-dire celui minimisant la fonction de

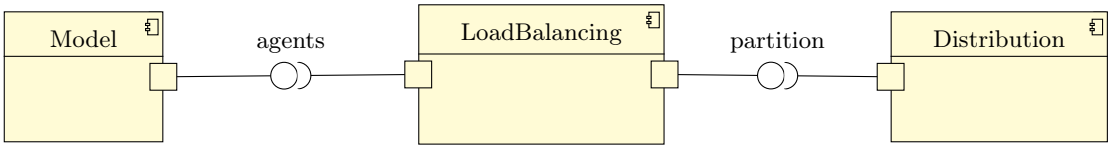


FIGURE 4.1. – Diagramme de composants UML pour l'interface d'équilibrage de charge

coût définie par l'équation 4.4. En effet, il est parfois nécessaire de prendre en compte d'autres critères comme le temps d'exécution de l'algorithme d'équilibrage de charge lui-même ou la difficulté d'implémentation et de paramétrage des algorithmes.

4.2.1. Spécification

Au chapitre précédent nous avons défini des méthodes permettant de distribuer une simulation tout en assurant la continuité et la cohérence des modèles distribués quelles que soient les partitions construites par les algorithmes d'équilibrage de charge. Il est donc possible de construire une interface d'équilibrage de charge indépendante du processus de distribution, comme représenté sur la figure 4.1. L'équilibrage de charge prend en entrée une liste d'agents, et produit une partition. L'algorithme de distribution permet ensuite de distribuer le modèle, quelle que soit la partition construite.

L'interface présentée sur la figure 4.2 définit une méthode d'équilibrage générique, `balance()`, qui prend en entrée deux paramètres.

Une liste d'agents *locaux* est d'abord spécifiée sur chaque processus. Il est possible d'appliquer l'algorithme seulement à un sous-ensemble d'agents, en laissant inchangé la localisation des autres. L'utilisation d'un type d'agent générique permet de définir à la fois des algorithmes applicables à tout modèle et d'autres tirant parti des spécificités des agents, comme leur position dans l'espace.

Le paramètre `mode`, identique sur tous les processus, permet de distinguer les méthodes de construction de partitions, *partitionnement* ou *repartitionnement*.

De manière générale, l'utilisation d'une interface permet de choisir librement l'implémentation de l'algorithme d'équilibrage de charge à utiliser pour chaque modèle sans altérer le reste du système. La définition de cette interface et de ses paramètres n'est qu'un choix permettant de formaliser les algorithmes présentés dans ce document. Il est cependant possible d'adapter les mêmes concepts aux choix du développeur et à l'environnement de développement, par exemple en définissant deux méthodes d'équilibrage distinctes plutôt qu'un paramètre `mode`. De telles considérations n'altèrent cependant pas la structure logique de l'architecture de simulation et des algorithmes d'équilibrage proposés.

4.2.2. Période d'application

Les appels à l'algorithme d'équilibrage de charge sur tous les agents du modèle sont gérés en interne par la plateforme. Certains algorithmes peuvent nécessiter une application récurrente à chaque pas de temps. C'est par exemple le cas avec RepastHPC

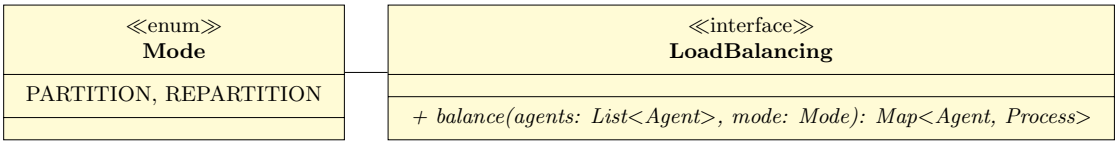


FIGURE 4.2. – Diagramme de classe de l'interface d'équilibrage de charge.

ou D-MASON, où il est nécessaire de migrer immédiatement les agents ayant franchis la frontière de la partie de l'environnement associée au processus courant.

En revanche, les algorithmes de continuité des données décrits au chapitre 3 nous permettent de lever ces contraintes. La cohérence du modèle étant garantie quelle que soit la distribution, il est possible d'exécuter des agents *locaux* n'étant plus assignés au même processus que la partie de l'environnement dans laquelle ils se trouvent, même si l'essentiel de leurs voisinages deviendront probablement *distants*.

Les algorithmes d'équilibrage peuvent donc s'appliquer à une période arbitraire et variable dans le temps, définie par l'utilisateur ou déterminée automatiquement selon l'évolution du modèle. De manière générale, en supposant que l'application de l'algorithme d'équilibrage aboutit à un meilleur équilibre, il est clair que la qualité des partitions d'un modèle croit lorsque la période d'application de l'équilibrage diminue. Cependant, il peut être préférable d'appliquer l'algorithme d'équilibrage à une fréquence plus faible, afin de trouver le meilleur compromis entre le temps d'exécution des algorithmes d'équilibrage de charge, qui implique un temps de migration des agents, et le gain induit par un meilleur équilibre de la simulation. Ce compromis dépend du modèle simulé et de l'environnement de simulation.

4.3. Modèles test

Plutôt que de tester les algorithmes d'équilibrages de charge implémentés à partir de l'interface générique sur des modèles spécifiques, nous introduisons différents types de modèles Multi-Agents, qui représentent eux-mêmes une approximation de certaines classes de modèles.

Dès lors l'implémentation des ces types de modèles dans un *Meta-Modèle* nous permet d'estimer les performances des algorithmes sur de larges classes de modèles.

4.3.1. Modèles à base de graphes purs

Les modèles à base de graphes purs représentent les modèles dans lesquels les agents interagissent entre eux selon un graphe, sans notion d'environnement ou de spatialité. C'est par exemple le cas dans la simulation de réseaux sociaux [23]. Le graphe d'interactions peut être statique ou dynamique.

Nous pouvons définir divers types de graphes statiques paramétrables, dont la structure présente des caractéristiques intéressantes ou assimilables à des modèles Multi-Agents existants. Les abréviations utilisées dans la suite pour désigner chaque type d'environnement sont données entre parenthèses.

Types de graphe

Grille (G). Une grille discrète régulière de taille $X \times Y$. Nous considérons que chaque cellule est connectée à ses 8 voisins, ce qui correspond au voisinage de Moore, même si d'autres schémas sont possibles. Bien que rarement utilisé en tant que graphe pur, ce type de graphe régulier permet de modéliser les interactions pouvant avoir lieu entre les cellules d'un environnement.

Graphe aléatoire (R, « Random »). Un graphe dans lequel chaque nœud est connecté à des voisins choisis aléatoirement. N nœuds sont d'abord initialisés. Le nombre de voisins sortants de chaque nœud est déterminé aléatoirement selon une loi de Poisson de paramètre K, de sorte que chaque nœud possède en moyenne K voisins sortants. Les voisins de chaque nœud sont ensuite choisis aléatoirement et uniformément parmi tous les autres nœuds. Ce graphe n'est pas le plus évident à associer à un modèle réel. Cependant, il pose un haut niveau de contrainte en termes de distribution. En effet, quelle que soit l'assignation des nœuds aux processus, chaque nœud a une forte probabilité de posséder des voisins sur d'autres processus, impliquant de nombreuses communications.

Graphe agrégé (C, « Clustered »). Ce graphe permet de représenter des modèles où les interactions ont lieu selon une notion de proximité spatiale. Pour le construire, N nœuds sont d'abord initialisés en leur assignant une position réelle $(x, y) \in [0, 1] \times [0, 1]$ selon une loi aléatoire uniforme. Comme précédemment, le nombre de voisins sortants est choisi selon une loi de Poisson de paramètre K. Les voisins de chaque nœud correspondent ensuite aux plus proches voisins, selon les coordonnées aléatoires précédentes. Il est important de noter que les coordonnées utilisées sont internes à l'algorithme, et oubliées par la suite.

Graphe Small World (SW). Ce type de graphe, entre le graphe aléatoire et le graphe agrégé, a été introduit par Watts et Strogatz [128] afin de mieux représenter les relations entre individus du monde réel [91]. La structure se retrouve également dans d'autres contextes, par exemple dans les réseaux de distribution d'énergie. La méthode de construction proposée par Watts et Strogatz étant conçue pour générer des graphes non orientés, nous adaptons la procédure de la manière suivante.

Nous supposons que le nombre moyen de voisins sortants, K, est pair. Un graphe cyclique avec N nœuds est d'abord initialisé, de sorte que chaque nœud soit connecté aux $K/2$ voisins le précédant dans le cycle, et au $K/2$ voisins le suivant. Un exemple est donné figure 4.3.

Chaque lien du graphe est ensuite sélectionné pour le relier avec une probabilité p. Pour chaque lien à relier, le côté à relier est choisi avec une probabilité $1/2$. Pour tous les liens choisis dont le nœud cible doit être relié, un nœud parmi tous ceux du graphe est choisi aléatoirement, et la cible du lien est changée pour ce nœud si et seulement si le nœud choisi n'est pas le nœud source, et s'il n'existe pas déjà un lien entre le nœud source et le nœud choisi. Les liens dont le nœud source doit être relié sont traités de manière analogue.

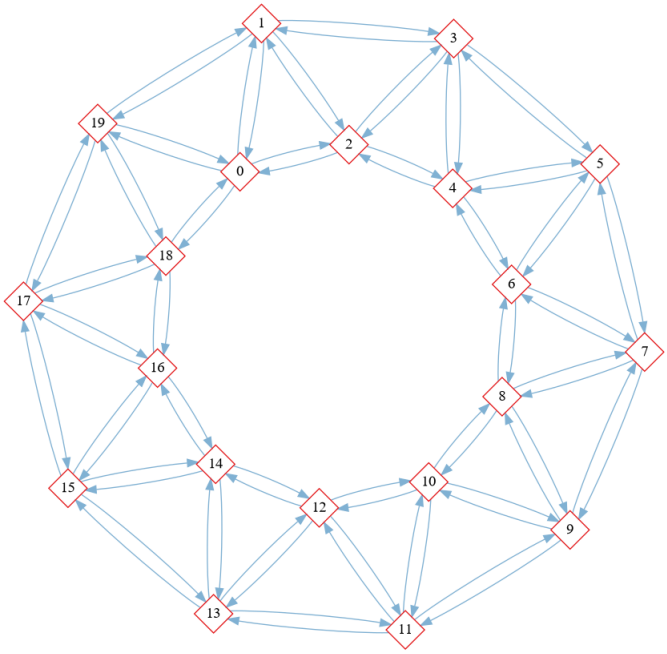


FIGURE 4.3. – Exemple de graphe cyclique à l'initialisation de la procédure de construction d'un graphe Small World avec $N = 20$ et $K = 4$.

Le graphe généré est parfaitement régulier pour $p = 0$, et complètement aléatoire pour $p = 1$.

Analyse

Le tableau 4.1 présente des exemples de caractéristiques obtenues pour différents types de graphes de taille comparable [1]. Dans le contexte de la génération aléatoire de graphes orientés de grande dimension, il est facilement possible d'obtenir des graphes non connectés, ou tout du moins faiblement connectés, avec des nœuds qui ne sont atteignables que dans un sens de parcours. C'est pourquoi, afin de valider la pertinence des graphes générés, nous ajoutons à C et L une valeur de *Connectivité*, définie comme le rapport entre la taille de la plus grande composante connectée et le nombre de nœuds dans le graphe.

On constate que la grille et le graphe agrégé ont des caractéristiques similaires. En effet, la grille peut-être vue comme un cas particulier de graphe agrégé, avec une très grande régularité. Ces deux types de graphes exposent un coefficient d'agrégation et une longueur de chemin caractéristique élevés par rapport au graphe aléatoire. En effet, la construction des voisinages par relation de proximité permet de fortement connecter les voisins entre eux à l'échelle locale, ce qui augmente en contrepartie le nombre de sauts nécessaires

1. Les valeurs ont été calculées grâce à l'outil *Python Graph-Tool* [8], à partir de graphes générés par FPMAS.

Environnement	C	L	Connectivité
Grille 1000×1000	0,429	467	100 %
Agrégé	0,557	364	99,6 %
Aléatoire	$7{,}50 \times 10^{-6}$	6,88	99,3 %
Small World ($p = 0{,}1$)	0,474	10,7	100 %

TABLE 4.1. – Valeurs des coefficients d'agrégation (C), de longueur de chemin caractéristique (L) et de Connectivité pour différents types d'environnements avec $N = 1\,000\,000$ et $K = 8$.

pour atteindre un nœud éloigné. En comparaison, le coefficient d'agrégation du graphe aléatoire est très faible car les voisins sont choisis parmi tout le graphe, indépendamment d'une quelconque notion de proximité, ce qui permet de se déplacer en un faible nombre de saut d'un bout à l'autre du graphe. Enfin, conformément à la définition de WATTS et STROGATZ, le graphe Small World présente à la fois un coefficient d'agrégation élevé et une faible longueur de chemin caractéristique. En effet, par construction, le graphe est initialement agrégé, puis la procédure de reconnection d'un nombre limité de liens permet de créer des raccourcis dans le graphe, réduisant la longueur des chemins, tout en limitant la décomposition des voisinages à l'échelle locale.

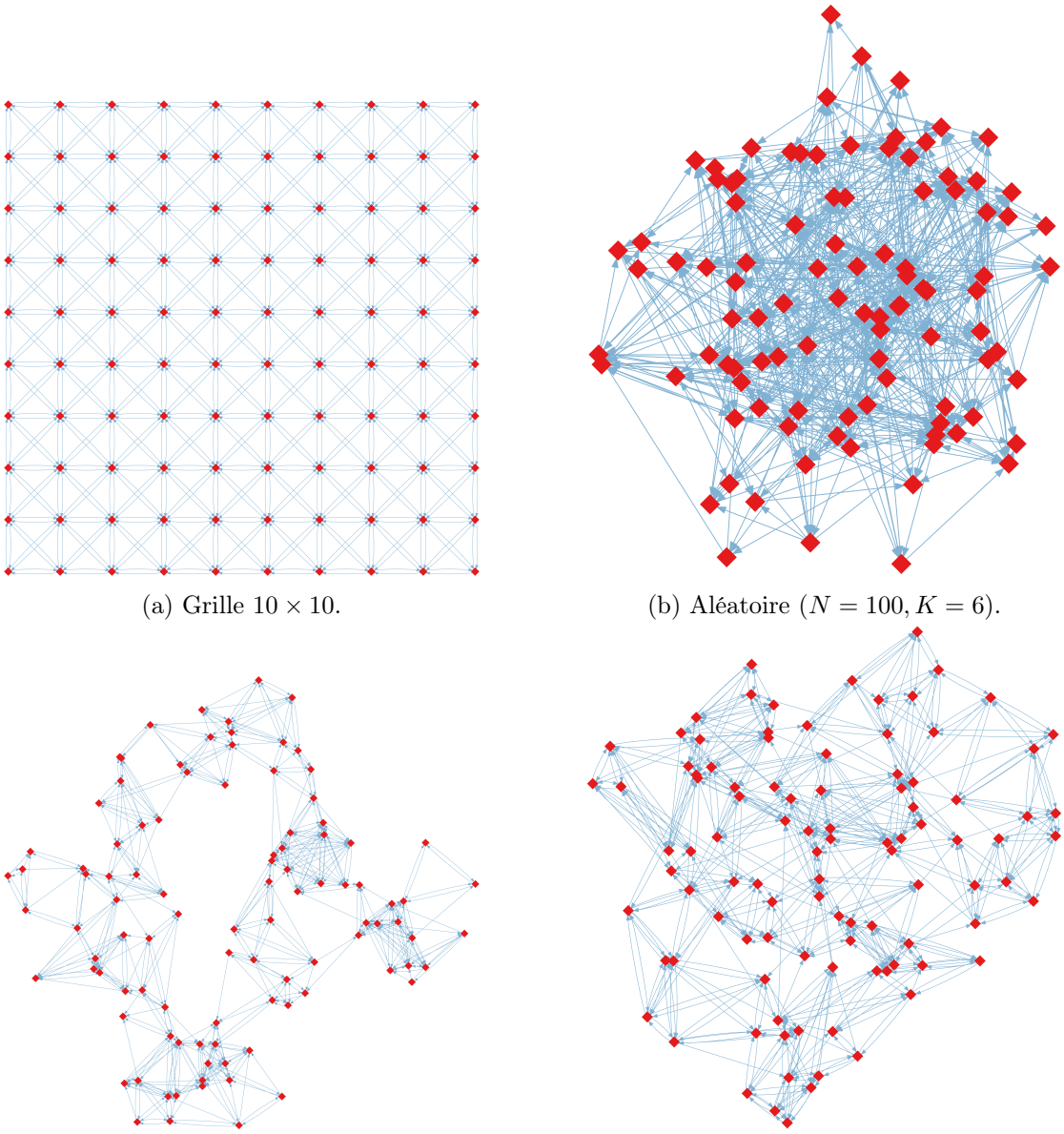
La figure 4.4 représente des exemples de graphes générés automatiquement grâce à FPMAS. Les méthodes de construction décrites ci-dessus étant séquentielles, elles ont été implémentées de manière distribuée dans FPMAS afin de bénéficier de l'exécution parallèle pour générer des graphes de grande taille et répartir leur utilisation de mémoire sur les nœuds de calcul.

4.3.2. Modèles spatiaux uniformes

Un moyen d'ajouter du dynamisme aux modèles consiste à utiliser les graphes précédents comme des environnements sur lesquels des agents se déplacent pour définir des modèles spatiaux. Dans ce type de modèle, les agents sont localisés dans une cellule de l'environnement. Ils peuvent alors se déplacer dans les cellules voisines de leur localisation, et y percevoir d'autres agents, selon leurs champs de déplacement et de perception. D'autres modèles pourraient considérer d'autres types d'interactions avec des agents éloignés spatialement. Ces interactions ne sont pas considérées dans notre étude, mais se rapprochent du cas du modèle spatial basé sur un environnement de type Small-World, où les agents au bout des raccourcis sont considérés comme spatialement proches même s'ils se situent éloignés dans le graphe initial. À noter qu'il n'est pas nécessaire d'associer des coordonnées explicites à chaque cellule pour définir un modèle spatial.

Nous considérons des exemples où la structure de l'environnement est statique. Le déplacement des agents induit cependant un dynamisme à l'échelle du modèle, avec notamment la mise à jour des perceptions des agents.

Le taux d'occupation de l'environnement, défini comme le rapport entre le nombre d'agents et le nombre de cellules, est un paramètre important des modèles spatiaux. Un



(a) Grille 10×10.

(b) Aléatoire ($N = 100, K = 6$).

(c) Agrégé ($N = 100, K = 6$).

(d) Small World ($N = 100, K = 6, p = 0, 1$).

FIGURE 4.4. – Exemple de graphes générés avec FPMAS.

modèle à base de graphe pur peut être conçu comme un modèle spatial avec un taux d'occupation nul.

Nous qualifions un modèle spatial d'uniforme lorsque la distribution des agents sur les cellules reste uniforme au cours du temps. C'est notamment le cas dans un modèle où les agents sont initialisés aléatoirement et uniformément sur le réseau de cellules, puis se déplacent aléatoirement à chaque pas de temps vers une des cellules voisines, en supposant négligeable la partie faiblement connectées du graphe orienté représentant les cellules, dans laquelle les agents pourraient se retrouver piégés. Dans la pratique, l'existence possible de nœuds qui n'appartiennent pas à la composante fortement connectée des graphes aléatoires ou agrégés présentés dans le tableau 4.1 n'est pas un problème. En effet, par construction, ces nœuds ne possèdent que des liens sortant vers la plus grande composante fortement connectée du graphe. Il est donc garanti que les agents sortent rapidement de ces quelques nœuds, sans pouvoir y entrer à nouveau.

Les version NetLogo des modèles Proie-Prédateur [14] et Virus [13] sont des exemples de modèles spatiaux uniformes à base de grille.

4.3.3. Modèles spatiaux non uniformes

Dans d'autres modèles, la répartition des agents sur l'environnement ne reste pas uniforme au cours du temps. C'est par exemple le cas avec les modèles de nuées d'oiseaux [11], le modèle Sugarscape [53], ou les modèles de colonies de fourmis où les agents se concentrent sur les zones riches en nourriture.

Afin de générer des modèles spatiaux non uniformes génériques, il est possible d'assigner une valeur d'utilité aux cellules de l'environnement. Les agents sont initialisés uniformément sur l'environnement. Leur comportement consiste ensuite à se déplacer aléatoirement dans une cellule voisine de sorte que la probabilité de choisir une cellule soit proportionnelle à son utilité. Pour générer un modèle spatial uniforme à partir de ce comportement, il suffit d'attribuer la même utilité à toutes les cellules.

Pour des raisons de simplicité, nous limitons la conception de fonctions d'utilité non uniformes aux modèles à base de grille. La méthode consiste d'abord à définir un ou plusieurs centres d'attraction, puis à associer une utilité aux cellules en fonction de leur distance au centre d'attraction. Si plusieurs centres d'attraction sont utilisés, l'utilité est définie comme la somme des utilités générées par chaque centre. Des fonctions d'utilité basiques sont présentées sur la figure 4.5

Diverses expérimentations nous ont cependant conduit à définir l'utilité *Marche-Inverse*, de sorte que l'utilité soit une *marche* de hauteur 1 000 pour les distances inférieures au rayon, et égale à l'utilité *inverse* de la figure 4.5 au delà. Cette utilité permet aux agents de se déplacer progressivement jusqu'au disque de haute utilité au centre, puis de s'y déplacer uniformément sans en sortir.

4.3.4. Meta-Modèle

Plutôt que de présenter des expériences difficilement généralisables basées sur des exemples particuliers, nous essayons d'identifier des classes de modèles et de fournir

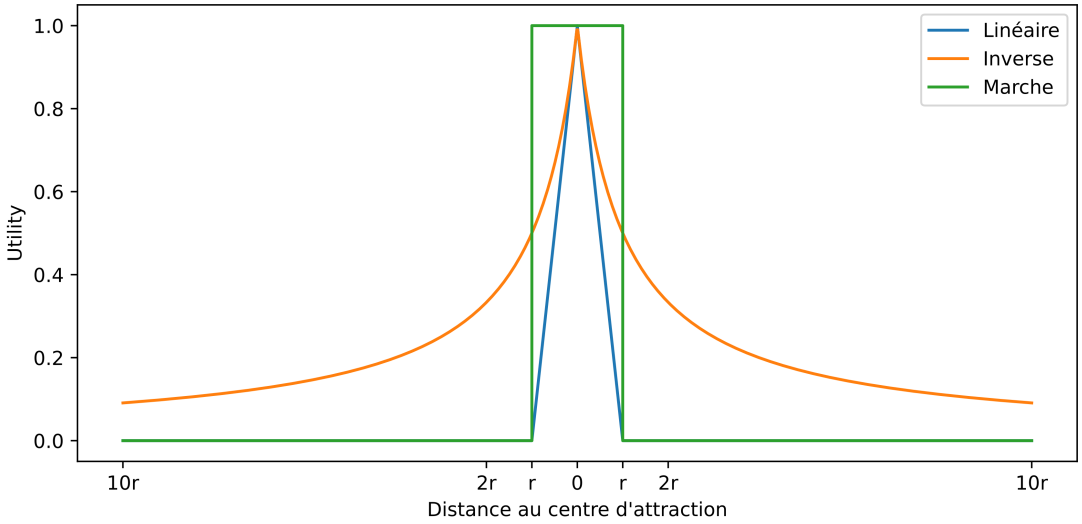


FIGURE 4.5. – Exemples de fonctions d'utilité basiques.

Modèle Multi-Agents	Type de modèle	Type de graphe
Véhicules sur un graphe urbain	Spatial uniforme	Graphe agrégé
Virus Proie-Prédateur	Spatial uniforme	Grille
Colonies de fourmis	Spatial non-uniforme	Grille
Communications sur un réseau social	Graphe pur	Small-World

TABLE 4.2. – Assimilation de quelques modèles Multi-Agents à des configurations du *Méta-Modèle*.

des résultats expérimentaux pertinents pour de nombreux modèles.

Nos expérimentations se basent donc sur la définition d'un *Meta-Modèle* paramétrable dont le but n'est pas d'illustrer un comportement particulier mais d'imiter des dynamiques rencontrées dans de nombreux modèles et pertinentes dans le contexte de la distribution de simulation de SMA. Le *Meta-Modèle* permet notamment de simuler les classes de modèles précédemment définies.

Le principe consiste ainsi, pour prévoir les performances de chaque algorithme d'équilibrage sur un modèle donné, à assimiler ce modèle à un cas du *Méta-Modèle* pour lequel les résultats sont disponibles. Ce travail d'assimilation est laissé au lecteur. Quelques exemples sont cités dans le tableau 4.2 pour illustrer le principe.

L'assimilation ne se fait pour le moment que sur la structure de l'environnement et le schéma de déplacement des agents. En effet, le temps d'exécution réel des agents n'a aucun impact sur les algorithmes d'équilibrage considérés. Même les algorithmes de partitionnement de graphe se basent sur le poids des agents, dont la valeur peut être

définie indépendamment du comportement réel des agents. Les agents et les cellules du *Méta-Modèle* ne sont donc pour le moment pas associés à un comportement réel. Dans le chapitre 5, une méthode d'assimilation des comportements est proposée afin d'évaluer les performances des différents schémas de communications.

Le seul comportement des agents consiste ici à se déplacer aléatoirement sur les cellules selon un champ de perception de taille 1. Dans le cas de l'environnement de type grille, les agents se déplacent dans le voisinage de Moore. Pour les autres graphes, les agents choisissent une destination parmi les voisins de la cellule dans laquelle ils sont localisés. Nous supposons la présence d'une barrière de synchronisation entre l'exécution des agents et des cellules. Plus précisément, l'exécution d'un pas de temps peut être décrite comme suit, en supposant une barrière de synchronisation stricte à chaque étape :

— étape 1 : exécution des agents ;
— étape 2 : exécution des cellules ;
— étape 3 : exécution de l'algorithme d'équilibrage de charge, selon sa fréquence d'exécution, et distribution du modèle selon la nouvelle partition.

Aucune interaction n'a réellement lieu dans le modèle, autre que la mise à jour des champs de perception et de déplacement. Ainsi, plutôt que de mesurer le temps d'exécution de ces agents et cellules fictifs, nous mesurons, à chaque pas de temps et sur chaque processus, les grandeurs suivantes :

1. T_{lb} : temps d'exécution de l'algorithme d'équilibrage de charge ;
2. T_{dist} : temps passé à distribuer le modèle (mise en place de la nouvelle partition) ;
3. W_a : poids total des agents sur le processus ;
4. W_c : poids total des cellules sur le processus ;
5. $L_{a\to a}$: poids total des liens depuis des agents *locaux* vers des agents *délégués* ;
6. $L_{c\to c}$: poids total des liens depuis des cellules *locales* vers des cellules *déléguées* ;
7. $L_{a\to c}$: poids total des liens depuis des agents *locaux* vers des cellules *déléguées*.

Seuls T_{lb} et T_{dist} représentent des temps d'exécution réels. Dans l'hypothèse où tous les poids sont unitaires, les autres grandeurs correspondent respectivement au nombre d'agents ou de liens associés. Nous ne considérons pas les liens entre deux entités *locales* pour deux raisons :

1. Le coût associé aux interactions est négligeable par rapport à celui des interactions distantes.
2. Leur coût peut être inclus dans les temps d'exécution du comportement des agents et des cellules.

Lors d'un pas de temps où l'application de l'algorithme d'équilibrage n'a pas lieu, nous obtenons naturellement $T_{lb} = T_{dist} = 0$.

On constate que toutes ces grandeurs permettent d'évaluer empiriquement le coût de l'équilibrage de charge dynamique défini à l'équation 4.4, en mettant en place l'égalité

suivante :

$$\begin{aligned} C_{\mathcal{S}',\mathcal{S}} &= \alpha(C_{\mathcal{S}}^{W} + C_{\mathcal{S}}^{L}) + C_{\mathcal{S}',\mathcal{S}}^{M} \\ &= \alpha(\sum_{p\in P} |W_a - W_a^*| + \sum_{p\in P} |W_c - W_c^*|) \\ &\quad + \alpha \sum_{p\in P} (L_{a\to a} + L_{a\to c} + L_{c\to c}) \\ &\quad + \max_{p\in P} (T_{dist}) \end{aligned} \tag{4.5}$$

Les coûts en charge de calcul pour les agents et les cellules sont calculés indépendamment, en raison de la barrière de synchronisation entre l'exécution des agents et des cellules. Le coût en communication est définit comme le total des poids des liens entre tous les types d'agents, qui représentent les interactions. Le coût de migration, qui représente le temps de distribution réel, est défini comme le maximum observé sur tous les processus, afin d'obtenir une borne supérieure du coût de migration.

Nous pourrions donc utiliser la fonction de coût 4.5 pour évaluer empiriquement les performances de chaque algorithme d'équilibrage à partir des résultats du *Meta-Modèle*. Cependant, afin d'assimiler plus facilement les résultats du *Méta-Modèle* à des modèles réels, nous proposons une méthode d'estimation des temps d'exécution du *Méta-Modèle*.

Les paramètres suivants permettent de représenter le comportement réel des agents dans le *Méta-Modèle* :

- t_a : estimation du temps d'exécution d'un agent ;
- t_c : estimation du temps d'exécution d'une cellule ;
- $t_{a\to a}$: estimation du temps de communication selon un lien entre un agent *local* et un agent *distant* ;
- $t_{a\to c}$: estimation du temps de communication selon un lien entre un agent *local* et une cellule *distante* ;
- $t_{c\to c}$: estimation du temps de communication selon un lien entre une cellule *locale* et une cellule *distante*.

Nous pouvons alors estimer le temps d'exécution des agents et des cellules sur le processus P grâce aux formules suivantes :

$$T_{a,P} = t_a W_a + t_{a\to a} L_{a\to a} + t_{a\to c} L_{a\to c} \tag{4.6}$$

$$T_{c,P} = t_c W_c + t_{c\to c} L_{c\to c} \tag{4.7}$$

Pour rappel, les temps d'équilibrage de charge et de distribution sont des temps réels mesurés sur chaque processus. De plus, nous supposons une barrière de synchronisation stricte entre les étapes d'exécution d'un pas de temps précédemment définies. Dès lors, nous pouvons estimer le temps d'exécution d'un pas de temps du modèle grâce à la formule suivante :

$$T = \max_{p\in P}(T_{a,P}) + \max_{p\in P}(T_{c,P}) + \max_{p\in P}(T_{lb} + T_{dist}) \tag{4.8}$$

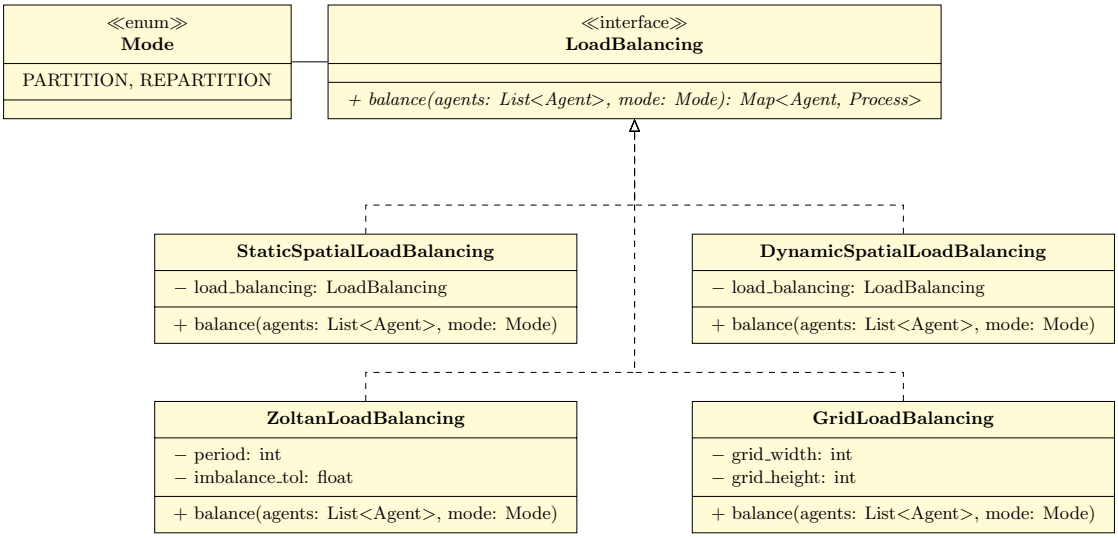


FIGURE 4.6. – Diagramme de classes présentant quelques algorithmes d'équilibrage de charge.

Le maximum associé à la troisième étape est calculé sur la somme de T_{lb} et T_{dist}, car nous ne supposons pas de barrière de synchronisation stricte entre l'équilibrage et la distribution du modèle.

Une discussion sur la méthode d'estimation des coûts de communication du *Méta-Modèle* est disponible en annexe B.

Le temps d'exécution total estimé du modèle est facilement obtenu par la somme des temps estimés pour chaque pas de temps.

4.4. Algorithmes d'équilibrage

Il est possible, à partir de l'interface d'équilibrage, de définir plusieurs algorithmes d'équilibrage de charge. Dans les sections suivantes, nous décrivons l'implémentation possible des algorithmes présentés sur la figure 4.6.

4.4.1. Équilibrage de charge à base de graphe

Une méthode d'équilibrage de charge de simulation de SMA consiste à partitionner l'ensemble des agents grâce à un algorithme de partitionnement de graphe. Des librairies de partitionnement de graphe et leur applicabilité à la simulation de SMA ont déjà été évoquées à la section 2.7.2.La méthode est évidente pour le cas d'un modèle à base de graphe. De manière générale, il suffit de construire un graphe avec les agents pour nœuds et des liens pour chaque interaction potentielle pour appliquer un algorithme de partitionnement de graphe à tout modèle Multi-Agents, indépendamment de la structure de données utilisée par le simulateur. Le choix d'un graphe comme structure de données

permet cependant d'appliquer nativement les algorithmes de partitionnement de graphe à nos modèles.

Notre implémentation de l'équilibrage de charge à base de graphe dans FPMAS se base sur la librairie Zoltan, pour des raisons de performance, de qualité d'implémentation et de support du repartitionnement, et pour l'implémentation distribuée de la librairie.

Conversion du modèle Multi-Agents vers Zoltan

L'algorithme 8 présente les fonctions qu'il est nécessaire de fournir à Zoltan pour :
— lister les nœuds ;
— obtenir le poids de chaque nœud ;
— lister les liens ;
— obtenir le poids de chaque lien ;
— définir le coût de migration de chaque nœud.

La construction des nœuds (lignes 1 et 4) ne pose pas de problème spécifique. Le poids des nœuds n'a pas a être spécifié dans la même unité que le poids des liens. En revanche, le coût de migration doit être spécifié dans la même unité que les coûts de communication. Le coût de migration d'un nœud (ligne 23) est obtenu par la somme du coût de migration de l'agent et des coûts de migration de ses voisins. En effet, conformément aux algorithmes de migration définis au chapitre 3, la migration d'un agent implique la migration d'une copie de son voisinage afin de maintenir la continuité des données. Dans le cas de la représentation à base de graphe, le coût de migration d'un voisin représente le coût de migration de tous les liens entre l'agent et ses voisins. La conversion détaillée d'un modèle Multi-Agents vers la structure d'hypergraphe de Zoltan est discutée dans l'annexe C.

Implémentation de l'interface d'équilibrage

La librairie Zoltan et son algorithme de partitionnement de graphe dynamique peuvent être utilisés pour implémenter une méthode d'équilibrage de SMA comme présenté dans l'algorithme 9.

La méthode d'initialisation doit être appelée une fois, avant le début de la simulation. Dans un contexte de programmation orientée objet, ces instructions peuvent typiquement être appelées au niveau d'un constructeur. Parmi les nombreux paramètres proposés par Zoltan, nous discutons ici seulement des paramètres `IMBALANCE_TOL` et `PHG_REPART_MULTIPLIER`, dont la définition à des valeurs autres que celles fournies par défaut s'est avérée pertinente dans notre application de Zoltan à la simulation distribuée de SMA.

Le paramètre `IMBALANCE_TOL` désigne le taux de déséquilibre maximal de la partition construite. Ce paramètre est défini par les concepteurs de Zoltan comme le rapport entre la charge maximale pouvant être associée à un processus et la charge moyenne. Zoltan ne construit que des partitions respectant ce critère, et cherche à minimiser les communications parmi les solutions possibles. Un plus haut taux de déséquilibre maximal permet plus de flexibilité, mais peut produire des résultats inattendus. Par exemple, supposons le partitionnement d'un million de nœuds de poids unitaire sur

Algorithme 8 Fonction de requêtes à fournir à Zoltan.

```
1: algorithme ListeNoeuds(Agents)
2:     retourner liste des agents locaux.
3: fin algorithme

4: algorithme Poids(Noeud)
5:     retourner coût en calcul relatif de l'agent (1.0 par défaut)
6: fin algorithme

7: algorithme ListeLiens(Agents)
8:     Initialiser une liste de liens
9:     pour chaque agent local faire
10:        pour chaque voisin des voisinages de l'agent faire
11:            si voisin ∈ liste d'agents à équilibrer alors
12:                si aucun lien n'existe entre l'agent et le voisin alors
13:                    Ajouter à la liste un lien entre l'agent et le voisin
14:                fin si
15:            fin si
16:        fin pour
17:    fin pour
18:    retourner liste de liens
19: fin algorithme

20: algorithme Poids(Lien)
21:    retourner somme des coûts en communication relatifs pour chaque voisinage
       entre les deux agents (1.0 par voisinage par défaut)
22: fin algorithme

23: algorithme CoutMigration(Noeud)
24:    Initialiser le coût au coût de migration de l'agent (1.0 par défaut)
25:    pour chaque voisin du noeud faire
26:        Ajouter le coût de migration de l'agent voisin
27:    fin pour
28:    retourner coût total
29: fin algorithme
```

Algorithme 9 Équilibrage de charge basé sur Zoltan.

Entrée:

1: période d'application de l'algorithme (*period*)
2: taux de déséquilibre maximal (*imbalance_tol*)

3: **algorithme** INITIALISER
4: Configurer Zoltan avec les fonctions de requêtes de l'algorithme 8
5: SETZOLTANPARAM(IMBALANCE_TOL, *imbalance_tol*)
6: SETZOLTANPARAM(PHG_REPART_MULTIPLIER, $10 \times$ *period*)
7: **fin algorithme**
8: **algorithme** BALANCE(agents, mode)
9: SETZOLTANPARAM(LB_APPROACH, mode)
10: **retourner** Résultat de Zoltan appliqué aux agents
11: **fin algorithme**

64 processus. La charge moyenne vaut $1\,000\,000/64 = 15\,625$. Supposons ensuite un partitionnement qui n'associe aucun nœud à 4 processus, et $1\,000\,000/60 \simeq 16\,667$ nœuds aux 60 processus restants. Le taux de déséquilibre vaut alors $16\,667/15\,625 \simeq 1{,}06$. Dès lors ce partitionnement est acceptable pour un taux de déséquilibre de 1,1 (10 %), même s'il n'utilise pas 4 des 64 processus disponibles, mais pas pour un taux de déséquilibre de 1,01 (1 %). Or n'associer aucun nœud à plusieurs processus permet généralement de réduire les communications, Zoltan a donc tendance à favoriser ces solutions, dans la limite du possible. La définition du paramètre `IMBALANCE_TOL` est donc importante pour respecter des contraintes strictes en termes d'utilisation des processus. Comme nous le verrons dans les expérimentations Zoltan fournit cependant de bons résultats avec le taux de déséquilibre par défaut de 1,1.

Zoltan ne considérant que les solutions dont la charge est équilibrée conformément au paramètre `IMBALANCE_TOL`, ses concepteurs considèrent négligeable le coût $\alpha C_{\mathcal{S}}^{W}$ de la fonction de coût 4.4. Zoltan cherche donc à minimiser la quantité $\alpha C_{\mathcal{S}}^{L} + C_{\mathcal{S}',\mathcal{S}}^{M}$, où α correspond au paramètre `PHG_REPART_MULTIPLIER` de la librairie Zoltan [35]. Selon les auteurs, le paramètre α doit indiquer combien d'itérations de la simulation sont effectuées entre chaque équilibrage de charge. Cependant, nous avons observé que dans le cas $\alpha = 1$, le coût de migration défini dans l'algorithme 8 est tel que le coût de migration d'un agent, d'une valeur par défaut de $1{,}0 + N_{voisins}$, compense systématiquement le gain en coût de communication sur une itération, d'une valeur par défaut inférieure à $N_{voisins}$, ce qui correspond à la réduction maximale du nombre de relations avec des voisins *distants*. Donc même en appliquant Zoltan à chaque itération, Zoltan ne peut migrer aucun agent même en mode `REPARTITION`. Ce paradoxe a été résolu en appliquant un facteur multiplicateur défini empiriquement à une valeur de 10 à la période d'application de Zoltan dans la définition du paramètre `PHG_REPART_MULTIPLIER`.

La méthode d'équilibrage permet de configurer le mode d'application de Zoltan, `PARTITION` ou `REPARTITION`, en accord avec nos propres définitions.

4.4.2. Équilibrage de charge spatialisé statique

Dans le cas d'un modèle spatial, l'algorithme d'équilibrage de charge à base de graphe précédent s'applique naïvement à la totalité du graphe, sans distinction pour les agents et les cellules. L'objectif de l'équilibrage de charge spatialisé consiste à utiliser un algorithme d'équilibrage de charge existant, tout en tirant partie du fait que les agents sont localisés dans des cellules, et qu'ils ont tendance à interagir avec les agents des cellules voisines.

L'algorithme 10 montre comment implémenter l'équilibrage de charge spatialisé à partir de n'importe quel algorithme d'équilibrage fournissant l'interface présentée à la figure 4.2. L'algorithme d'équilibrage existant n'est appliqué qu'en mode `PARTITION` et seulement aux cellules. Le coeur de l'algorithme se situe à la ligne 8, où la localisation de chaque agent est systématiquement définie comme celle de la cellule dans laquelle il est localisé, en mode `PARTITION` comme en `REPARTITION`. Cet algorithme constitue une généralisation de l'équilibrage réalisé à chaque pas de temps avec les zones de recouvrement de RepastHPC ou D-MASON.

Son utilisation peut s'avérer efficace lorsque le graphe de cellules est statique et que la répartition des agents dans l'environnement reste uniforme. En effet, dans ce cas, quel que soit le partitionnement initial de l'environnement, chaque sous-partie de l'environnement contient en moyenne le même nombre d'agents et est donc associée à une charge constante. Il suffit donc d'équilibrer la distribution de l'environnement en début de simulation, sans même prendre en compte la répartition initiale des agents. L'algorithme maintient alors un équilibre de qualité au cours de la simulation, tout en évitant l'application récurrente d'un algorithme d'équilibrage potentiellement coûteux en temps de calcul à la totalité du modèle.

Algorithme 10 Équilibrage de charge spatialisé statique.

Entrée:

1: Équilibrage de charge existant (LB)

```
2: algorithme BALANCE(agents, mode)
3:     dans le cas du mode PARTITION
4:         cellules ← ensemble des cellules parmi les agents
5:         partition ← LB.BALANCE(cellules, PARTITION)
6:     fin cas
7:     pour chaque agent spatial des agents faire
8:         partition[agent] ← partition[cellule de l'agent]
9:     fin pour
10:    retourner partition
11: fin algorithme
```

4.4.3. Équilibrage de charge spatialisé dynamique

Le principe de l'équilibrage de charge spatialisé dynamique est similaire au cas statique dans le sens où les agents sont toujours systématiquement associés au même processus que leur localisation, mais l'équilibrage de charge existant est cette fois appliqué au graphe de cellules à chaque itération, y compris en mode `REPARTITION`. Contrairement au cas statique, le poids des agents est localement pris en compte à chaque itération, en ajoutant temporairement au poids de chaque cellule le poids des agents qui y sont localisés. L'algorithme d'équilibrage existant n'est ainsi appliqué qu'aux cellules, mais prend en compte la localisation et le coût en calcul des agents, et le fait que la migration d'une cellule entraine la migration de tous les agents qui y sont localisés. L'équilibrage de charge spatialisé dynamique représente ainsi un gain en performances par rapport à l'application naïve de l'équilibrage de charge existant sur tout le graphe.

L'algorithme permet de maintenir un équilibre dans le cas d'un réseau de cellules dynamique, mais aussi de s'adapter à une répartition non uniforme et dynamique des agents sur l'environnement, contrairement au cas statique. Le fait d'appliquer systématiquement l'algorithme d'équilibrage aux cellules peut cependant impliquer un coût en temps de calcul significatif. Le choix de la version statique ou dynamique de l'algorithme d'équilibrage de charge spatialisé doit donc se faire en fonction des contraintes de chaque modèle.

Algorithme 11 Équilibrage de charge spatialisé dynamique.

Entrée:
1: Équilibrage de charge existant (LB)

```
2: algorithme BALANCE(agents, mode)
3:     cellules ← ensemble des cellules parmi les agents
4:     Sauvegarder le poids des cellules
5:     pour chaque cellule faire
6:         pour chaque agent localisé dans la cellule faire
7:             Ajouter le poids de l'agent au poids de la cellule
8:         fin pour
9:     fin pour
10:    partition ← LB.BALANCE(cellules, mode)
11:    pour chaque agent spatial des agents faire
12:        partition[agent] ← partition[cellule de l'agent]
13:    fin pour
14:    Restituer le poids des cellules
15:    retourner partition
16: fin algorithme
```

4.4.4. Équilibrage de charge à base de grille

L'équilibrage de charge à base de grille consiste à décomposer l'espace selon une grille de sorte à associer une zone continue de l'espace à chaque processus. Les agents ou cellules sont ensuite associés à un processus en fonction de la zone dans laquelle ils se trouvent. Cette méthode est classiquement appliquée aux environnements spatiaux discrets ou continus par les plateformes existantes, comme présenté au chapitre 2. Elle est cependant facilement généralisable aux graphes spatiaux si chaque nœud est associé à des coordonnées spatiales explicites.

Étant donné un ensemble de processus P sur lequel distribuer une simulation, le problème consiste à décomposer un environnement de taille $X \times Y$ selon une grille de taille $n \times p$ contenant $|P|$ cellules. Nous supposons que la taille de l'environnement et le nombre de processus sont arbitraires : nous ne nous limitons donc pas aux cas où $X \simeq Y$ et $|P|$ est une puissance de 2, comme souvent observé dans les exemples classiques.

Ainsi une méthode efficace de construction de la grille consiste à choisir n et p parmi les couples de diviseurs de $|P|$ de sorte que le rapport $\frac{n}{p}$ soit le plus proche possible de $\frac{X}{Y}$. Lorsque $X = Y$, il suffit de choisir le couple de diviseurs le plus proche de $\sqrt{|P|}$.

Un exemple complet d'implémentation est présenté dans l'algorithme 12. Le fonctionnement de l'algorithme est similaire à celui de l'équilibrage de charge spatialisé statique précédent. Les cellules sont ainsi partitionnées seulement en mode `PARTITION` (ligne 14) et chaque agent est implicitement assigné au même processus que la cellule dans laquelle il est localisé, dans les modes `PARTITION` et `REPARTITION` (ligne 20). La structure de grille discrète permet d'inclure des optimisations notables, le partitionnement des cellules et des agents se limitant à l'accès d'un élément dans un tableau.

Comme pour l'équilibrage de charge spatialisé statique, la qualité de l'équilibrage de cet algorithme est conditionnée à la répartition uniforme des agents dans l'espace. En effet, contrairement à l'équilibrage de charge spatialisé dynamique ou à l'équilibrage de charge à base de graphe, l'algorithme ne prend pas en compte le poids des agents et ne peut s'adapter à la surcharge d'un processus due à la concentration des agents sur une partie de l'environnement.

Algorithme 12 Algorithme d'équilibrage de charge à base de grille.

Entrée:

Taille de la grille $(X \times Y)$
Ensemble de processus (P)

algorithme INITIALISER
    Déterminer n et p, couple de diviseur de $|P|$ qui minimise la quantité $|\frac{n}{p} - \frac{X}{Y}|$.
    $(N_x, N_y) \leftarrow (X/n, Y/p)$ ▷ nombre de cellules par processus
    grille $\leftarrow$ tableau de processus de taille $n \times p$
    Associer un processus à chaque indice du tableau
fin algorithme

algorithme BALANCE(agents, modes)
    dans le cas du mode PARTITION
        pour chaque cellule des agents **faire**
            $(x, y) \leftarrow$ coordonnées de la cellule
            $(i, j) \leftarrow (x/N_x, y/N_y)$ ▷ division entière
            partition[cellule] $\leftarrow$ grille[i][j]
        fin pour
    fin cas
    pour chaque agent spatial des agents **faire**
        $(x, y) \leftarrow$ coordonnées de l'agent
        $(i, j) \leftarrow (x/N_x, y/N_y)$ ▷ division entière
        partition[cellule] $\leftarrow$ grille[i][j]
    fin pourretourner partition
fin algorithme

4.5. Performances et comparaisons

Dans cette section nous présentons des résultats expérimentaux permettant de comparer les performances des algorithmes d'équilibrage de charge introduits précédemment sur différents types de modèles, pour confirmer certaines suppositions et apporter d'autres éléments de comparaison. Les différentes configurations expérimentales et les temps d'exécution estimés des modèles sont obtenus avec FPMAS 1.6 [2] et la version 1.1 de l'implémentation du *Meta-Modèle* [3]. Ces expérimentations et les suivantes ont été menées sur le calculateur du Mésocentre de Calcul de Franche-Comté, dont la configuration est donnée dans le tableau 4.3. Les données brutes correspondant aux résultats du *Meta-modèle* sont accessibles librement et de manière pérenne grâce à la plateforme dat@UBFC [27].

4.5.1. Graphe pur

Afin d'évaluer la capacité de Zoltan à équilibrer différents types de graphes assimilables à divers modèles Multi-Agents sur un nombre arbitraire de processus, nous mesurons les performances de l'algorithme d'équilibrage à base de graphe sur un modèle sans agent autre que les cellules de l'environnement.

A part les barrières de synchronisation et l'exécution de l'algorithme d'équilibrage de charge, rien ne se passe dans la simulation du *Meta-Modèle* avec cette configuration : aucun déplacement d'agent n'ayant lieu, le graphe reste le même tout au long de la simulation. Il n'est donc pas pertinent d'envisager une application dynamique de Zoltan. En revanche, il est possible d'évaluer la qualité des partitions construites par Zoltan sur les différents types d'environnements.

La figure 4.7 présente le temps d'exécution estimé d'un modèle pour différents types d'environnements. Les valeurs sont obtenues par la somme sur l'ensemble des pas de temps des facteurs correspondants dans l'équation 4.8 [2], de sorte que la hauteur totale de chaque barre corresponde au temps d'exécution estimé de la simulation calculé grâce à l'équation 4.8 appliquée aux valeurs réellement mesurées. De manière plus pragmatique, il suffit de considérer que chaque bloc correspond au temps total estimé passé à réaliser l'action correspondante au cours de la simulation [3]. Les paramètres d'expérimentation sont donnés dans le tableau 4.4. L'environnement de type grille est de taille 1000×1000,

2. Par exemple, *Communication Cellule* $\rightarrow$ *Cellule* correspond à la somme des facteurs $t_{c \rightarrow c} L_{c \rightarrow c}$ maximisant à chaque pas de temps la somme correspondante dans l'équation 4.8.

3. Par exemple, le bloc *Comportement Cellules* correspond au temps total passé à exécuter le comportement des cellules.

Processeur	Intel(R) Xeon(R) CPU E5-2640 v3
Réseau	Infiniband Qlogic QDR 40G/s
OS	CentOS 6.10
MPI	OpenMPI 4

Table 4.3. – Configuration utilisée sur le Mésocentre de Calcul de Franche-Comté.

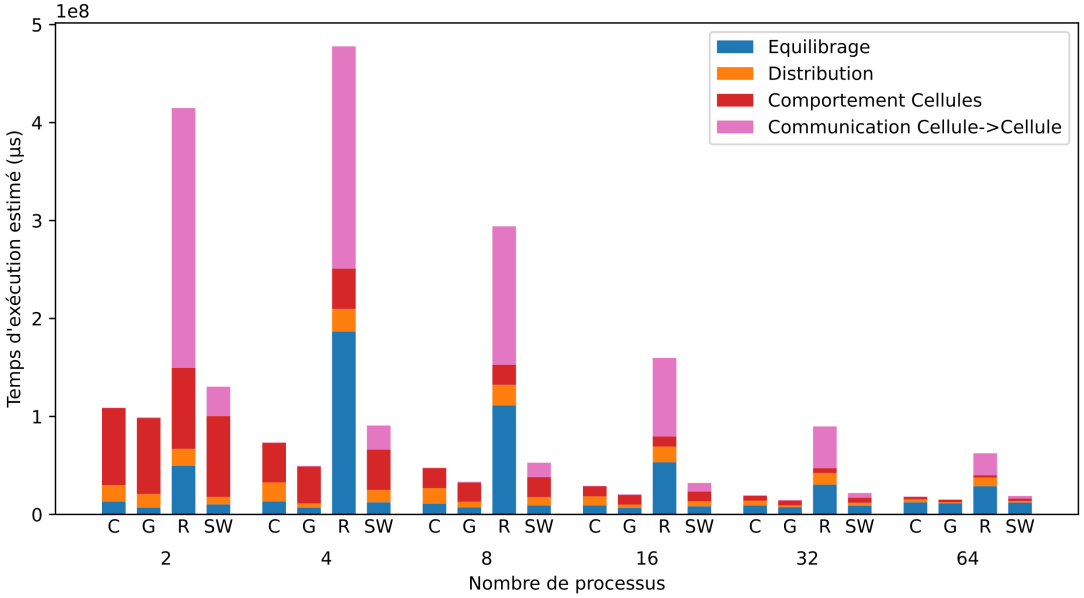


FIGURE 4.7. – Temps d'exécution estimés sur 10 pas de temps pour un modèle à base de graphe pur en appliquant Zoltan à différents types d'environnements.

de sorte que pour tous les environnements $N = 1\,000\,000$ et $K = 8$. Nous estimons le temps d'exécution de 10 pas de temps seulement, afin d'obtenir un résultat visuel. L'équilibrage de charge n'ayant lieu qu'une fois en début de simulation, les temps relatifs d'équilibrage de charge et de distribution deviennent négligeables au delà. Les résultats présentés sont obtenus par la moyenne de 10 expériences avec des graines différentes. Le coefficient de variation pour chaque temps total estimé est de l'ordre de 1%.

On observe tout d'abord que pour les environnements C, G et SW, le temps d'exécution de Zoltan est plutôt stable en fonction du nombre de processus. En effet, la complexité de l'algorithme croit en fonction du nombre de sous-partitions à construire, mais l'implémentation distribuée de Zoltan permet de compenser efficacement cette hausse de complexité. Quel que soit le nombre de processus, le temps d'exécution de Zoltan est plus élevé pour le graphe aléatoire R que pour les autres types de graphes. En effet, les autres font preuve d'un haut coefficient d'agrégation, ce qui simplifie la construction des partitions. Même sur ce cas le plus contraint, Zoltan permet d'obtenir de bonnes performances quand le nombre de processus augmente.

La nature du graphe aléatoire implique également des temps de communication plus élevés, les cellules voisines ayant plus de chance d'être localisées sur d'autres processus, caractéristique qui se retrouve dans une moindre mesure avec le graphe Small-World. Pour les graphes C et G, Zoltan permet de réduire les communications entres les processus au point de les rendre négligeables.

Il n'y a pas d'amélioration significative des performances globales de 32 à 64 processus, en raison d'un trop faible nombre moyen de nœuds par processus (31 250 pour 32

Paramètre	Valeur
N	1 000 000
K	8
$X \times Y$ (G)	1000×1000
p (SW)	0,1
$t_{cellule}$	15 µs
$t_{cellule \to cellule}$	20 µs
IMBALANCE_TOL	1,1
Nombre de pas de temps	10
Nombre d'expériences	10

TABLE 4.4. – Paramètres d'expérimentation pour le modèle à base de graphe pur.

processus et 15 625 pour 64 processus), rendant caduque la distribution.

On observe enfin que pour chaque nombre de processus, le temps d'exécution des cellules est le même quel que soit le type d'environnement, ce qui prouve la fiabilité de Zoltan dans l'équilibrage de la charge de calcul.

Ces expérimentations valident la capacité de Zoltan à équilibrer les différents types de modèles à base de graphes purs définis dans le *Méta-Modèle*. Dès lors, nous pouvons étudier le comportement de Zoltan sur un modèle dynamique.

4.5.2. Modèle spatial uniforme

Afin d'évaluer le comportement des algorithmes d'équilibrage de charge sur des modèles dynamiques, nous considérons maintenant le cas des modèles spatiaux avec une répartition uniforme des agents dans l'environnement. Nous utilisons Zoltan pour réaliser le partitionnement de l'environnement dans les algorithmes d'équilibrage spatialisés précédemment définis. Nos expérimentations se focalisent donc essentiellement sur l'utilisation de Zoltan et des variantes proposées sur des modèles Multi-Agents distribués. Seul l'environnement à base de grille permet d'utiliser le partitionnement à base de grille, qui ne dépend pas de Zoltan.

L'aspect dynamique des modèles simulés pose la question de la période d'application des algorithmes d'équilibrage. En effet, FPMAS permet l'application de tous les algorithmes d'équilibrage avec une période arbitraire. On constate que l'équilibrage spatialisé statique et l'équilibrage de charge à base de grille font preuve d'une faible complexité temporelle en mode `REPARTITION`, qui ne nécessite qu'un parcours de l'ensemble des agents spatiaux à partitionner. Il est donc raisonnable de considérer une application à chaque pas de temps de ces algorithmes. En revanche, les algorithmes impliquant une application dynamique de Zoltan possèdent une complexité non négligeable en mode `REPARTITION`. Nous utilisons donc une période de 10 pas de temps pour l'application de ces algorithmes.

La conception et l'analyse qualitative des algorithmes réalisées à la section 4.4 nous permettent de prédire en partie leurs performances relatives sur certains types

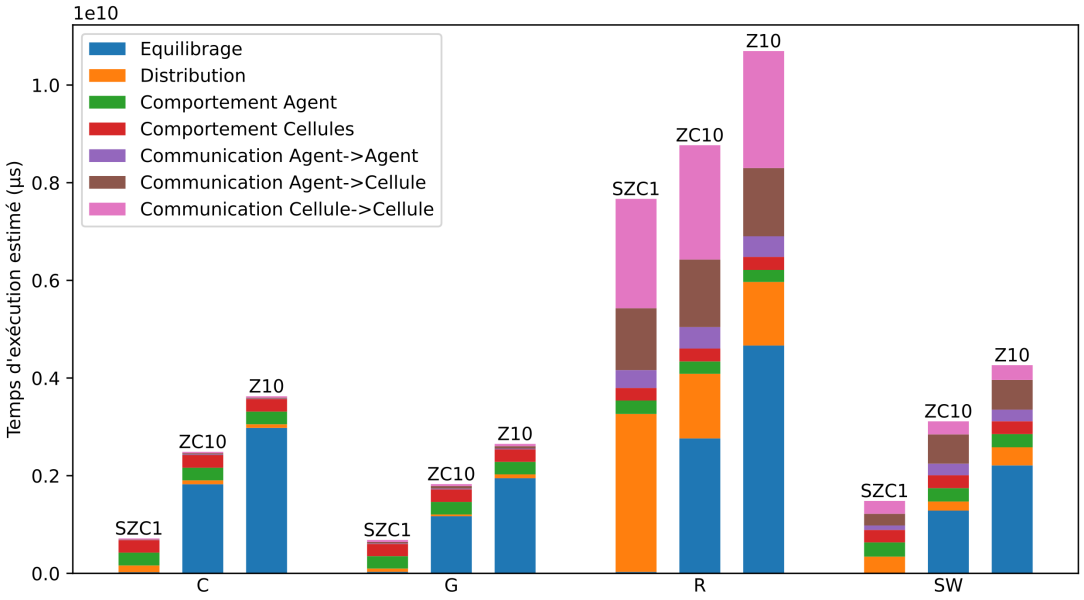


FIGURE 4.8. – Temps d'exécution estimé d'un modèle spatial uniforme pour différents types d'environnements sur 64 processus.

de modèles. Ainsi l'algorithme d'équilibrage spatialisé statique (SZC, « Static Zoltan Cell ») est théoriquement plus performant que l'équilibrage spatialisé dynamique (ZC, « Zoltan Cell ») et que l'équilibrage de charge à base de graphe (Z, « Zoltan ») dans le cas d'un modèle spatial uniforme. Pour vérifier cette hypothèse, nous effectuons la simulation d'un modèle spatial uniforme pendant 1000 pas de temps avec différents types d'environnements, à l'aide de 64 processus. Le détail des résultats est présenté sur la figure 4.8. La période d'application des algorithmes est rappelée par le nombre accolé à leurs abréviations. Les paramètres expérimentaux sont donnés dans le tableau 4.5. Les paramètres de temps individuels d'exécution et de communications sont relativement arbitraires : nous considérons dans cet exemple que le temps d'exécution des agents est deux fois supérieur à celui des cellules, que le temps de communication des agents est inférieur au temps d'exécution de leur comportement, et que le temps de communication entre les cellules est le même que celui des agents avec les cellules. Le *Méta-Modèle* permet par nature d'étudier facilement de nombreuses autres configurations, mais le détail des valeurs relatives n'a que peu d'impact sur les résultats observés, les ordres de grandeur des valeurs choisies étant justifiés au chapitre 5.

L'hypothèse sur les meilleures performances de l'algorithme SZC est largement confirmée dans ce cas. En effet, pour chaque environnement, la somme des temps d'exécution et de communication sont similaires : les trois algorithmes fournissent donc des partitions de qualité similaires. On observe en particulier que l'application tous les 10 pas de temps des algorithmes Z et ZC n'est pas limitante en termes de qualité des partitions, mais divise par 10 le nombre d'applications de Zoltan. Les performances de

Paramètre	Valeur
N	1 000 000
K	8
$X \times Y$ (G)	1000×1000
p (SW)	0,1
Taux d'occupation	0,5
$t_{cellule}$	15 µs
$t_{cellule \to cellule}$	20 µs
t_{agent}	30 µs
$t_{agent \to cellule}$	20 µs
$t_{agent \to agent}$	10 µs
IMBALANCE_TOL	1,1
Nombre de pas de temps	1000
Nombre d'expérience	10

TABLE 4.5. – Paramètres d'expérimentation pour le modèle à base de graphe pur.

ces algorithmes sont cependant largement dépassées par l'algorithme SZC, dont le temps d'équilibrage est négligeable par rapport aux deux autres.

Ainsi, pour des raisons de cohérence et de temps de calcul, chaque expérimentation pouvant durer plusieurs jours, nous ne présentons pas d'étude extensive des performances des algorithmes Z et ZC dans le cas des modèles spatiaux uniformes. Leur passage à l'échelle est tout de même garanti par les résultats de Zoltan dans le cas d'un graphe pur déjà présentés. En effet, si Zoltan est capable de construire des partitions de qualité sur un graphe statique, il est également capable d'en fournir un de qualité de manière itérative, les performances du repartitionnement de Zoltan étant déjà garanties par ses concepteurs.

Il est cependant nécessaire d'étudier le passage à l'échelle de la méthode SZC, pour s'assurer que le déplacement des agents sans application itérative de Zoltan n'induit pas de déséquilibre, quel que soit le nombre de processus. D'où les résultats présentés sur la figure 4.9. Ces expériences, réalisées avec les mêmes paramètres que les précédentes en faisant varier le nombre de processus, confirment les performances de la méthode SZC.

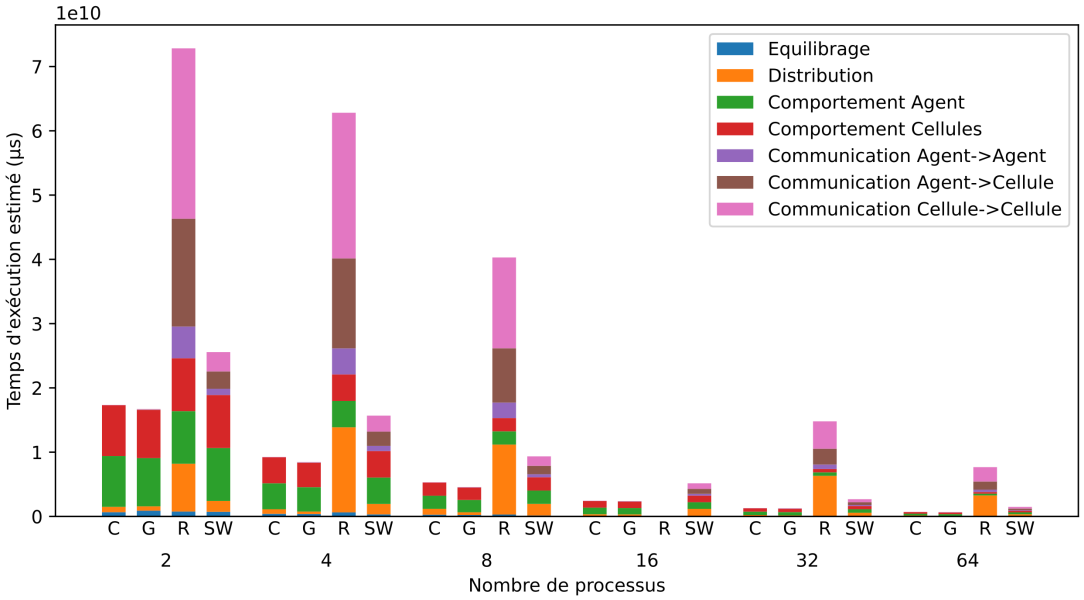


FIGURE 4.9. – Temps d'exécution estimé d'un modèle spatial avec l'algorithme d'équilibrage de charge spatialisé statique appliqué à chaque pas de temps (SZC1) pour différents types d'environnements.

4.5.3. Modèle spatial non uniforme

L'application dynamique de Zoltan trouve notamment son intérêt dans l'équilibrage de modèles avec une répartition d'agent non uniforme. La configuration du *Meta-Modèle* permet de générer ce type de modèles grâce à l'utilité des cellules et au comportement de déplacement des agents.

Nous définissons un modèle non uniforme grâce à une grille de taille 512×512, de sorte qu'une sous-partie de l'environnement de taille 64×64 soit associée à chaque processus lors d'une simulation avec 64 processus équilibrée grâce à l'équilibrage de charge à base de grille. Le partitionnement des cellules ainsi obtenu est présenté sur la figure 4.10.

Afin d'induire une répartition non uniforme et dynamique des agents, nous définissons l'utilité des cellules grâce à la fonction *Marche-Inverse* définie dans la section 4.3, le centre d'attraction étant situé au centre de la grille, avec un rayon de 150. Le modèle est simulé sur une durée de 10 000 pas de temps. L'évolution de la distribution des agents à plusieurs dates est présentée dans la figure 4.11.

Les agents, initialement répartis uniformément, se déplacent aléatoirement avec une légère attraction vers le disque central. Les agents dans le disque s'y déplacent uniformément, avec une probabilité négligeable d'en sortir. Dès lors le disque se remplit progressivement, au détriment du reste de l'environnement. Il est alors clair qu'avec le partitionnement à base de grille présenté dans la figure 4.10, les processus au centre font face à une surcharge de plus en plus importante.

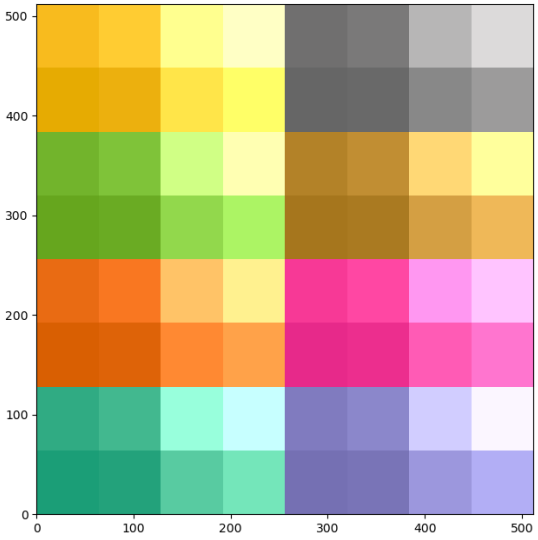


FIGURE 4.10. – Partitionnement sur 64 processus d'une grille discrète.

L'expérimentation consiste alors à observer la capacité de l'équilibrage de charge spatialisé dynamique basé sur Zoltan à adapter le partitionnement pour maintenir une charge équilibrée entre les processus. Des résultats sont présentés sur la figure 4.12. À $t = 0$, lorsque la répartition des agents est encore uniforme, Zoltan agrège les cellules en sous-partitions de tailles similaires. On constate ensuite que Zoltan a tendance à réduire la taille des sous-partitions au centre pour augmenter la taille des sous-partitions en périphérie, où la densité des agents est moindre, afin de conserver un nombre équilibré d'agents sur chaque processus.

Nos expérimentations ont cependant montré que même dans ce cas, le gain induit par un meilleur équilibrage ne compense pas systématiquement le temps d'exécution de Zoltan. En effet l'impact du déséquilibre est plus ou moins significatif selon les temps d'exécution et de communication des agents. Grâce au *Meta-Modèle*, il est possible d'évaluer précisément les points de bascule à partir desquels Zoltan améliore les performances globales du modèle.

La figure 4.13a présente une comparaison des temps d'exécution estimés du modèle non uniforme précédent en fonction du temps d'exécution des agents, après avoir fixé le temps de communication entre agent *distants* à 100 µs. On constate que pour des temps d'exécution par agent faibles, l'équilibrage à base de grille est plus performant que Zoltan. En effet, l'importance du déséquilibre décroit quand le temps d'exécution de chaque agent diminue, ce qui tend à favoriser l'équilibrage de charge à base de grille dans le compromis entre un meilleur équilibre et le temps d'exécution de l'équilibrage de charge. Ce compromis bascule en faveur de Zoltan au delà d'environ 350 µs de temps d'exécution par agent. Or, comme nous l'avons observé dans certaines de nos expérimentations, des temps d'exécutions inférieurs à 350 µs restent réalistes. Il existe donc des situations où l'équilibrage à base de grille statique est plus performant que Zoltan, y compris dans le

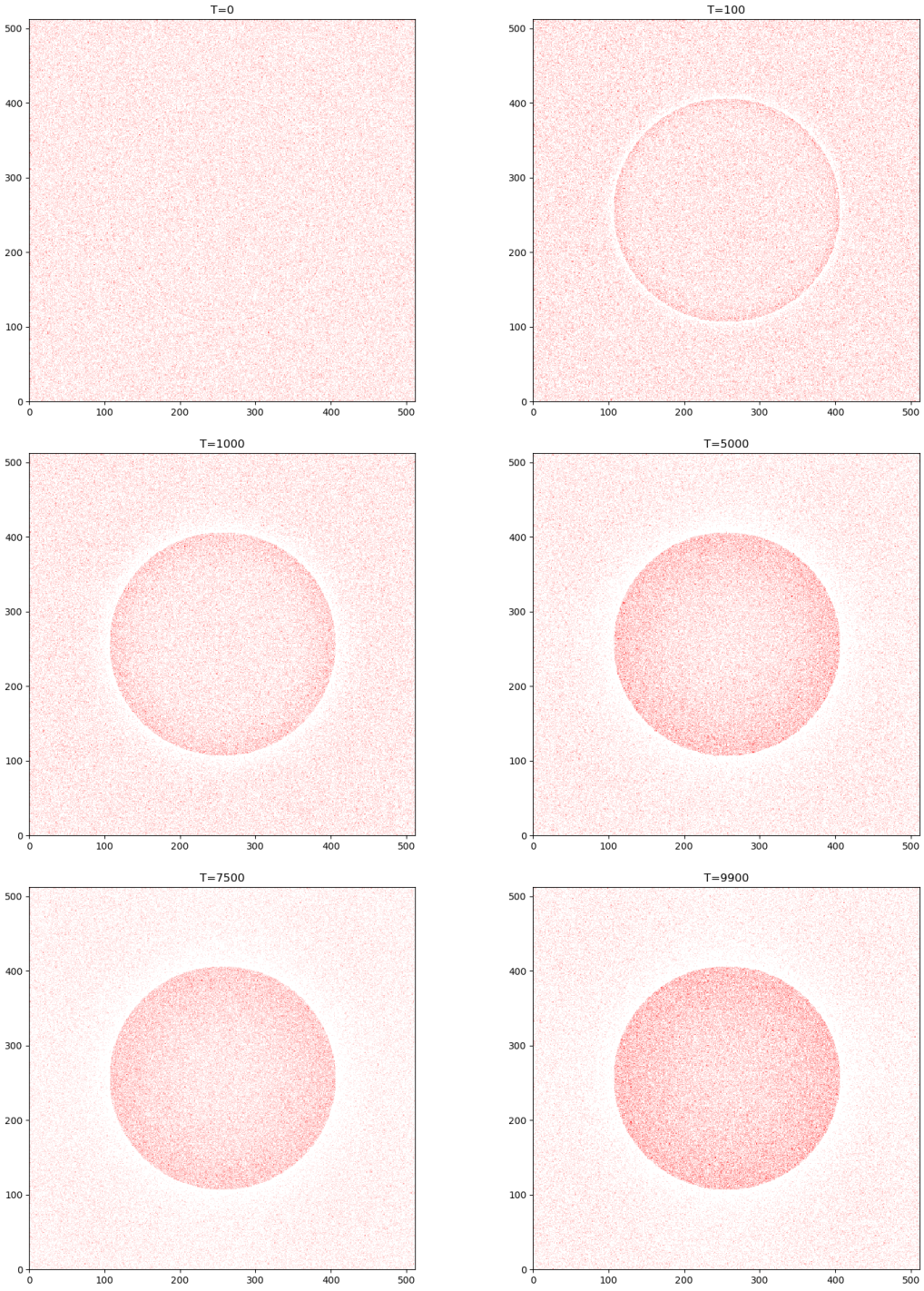


FIGURE 4.11. – Évolution temporelle de la densité des agents dans un modèle spatial non uniforme.

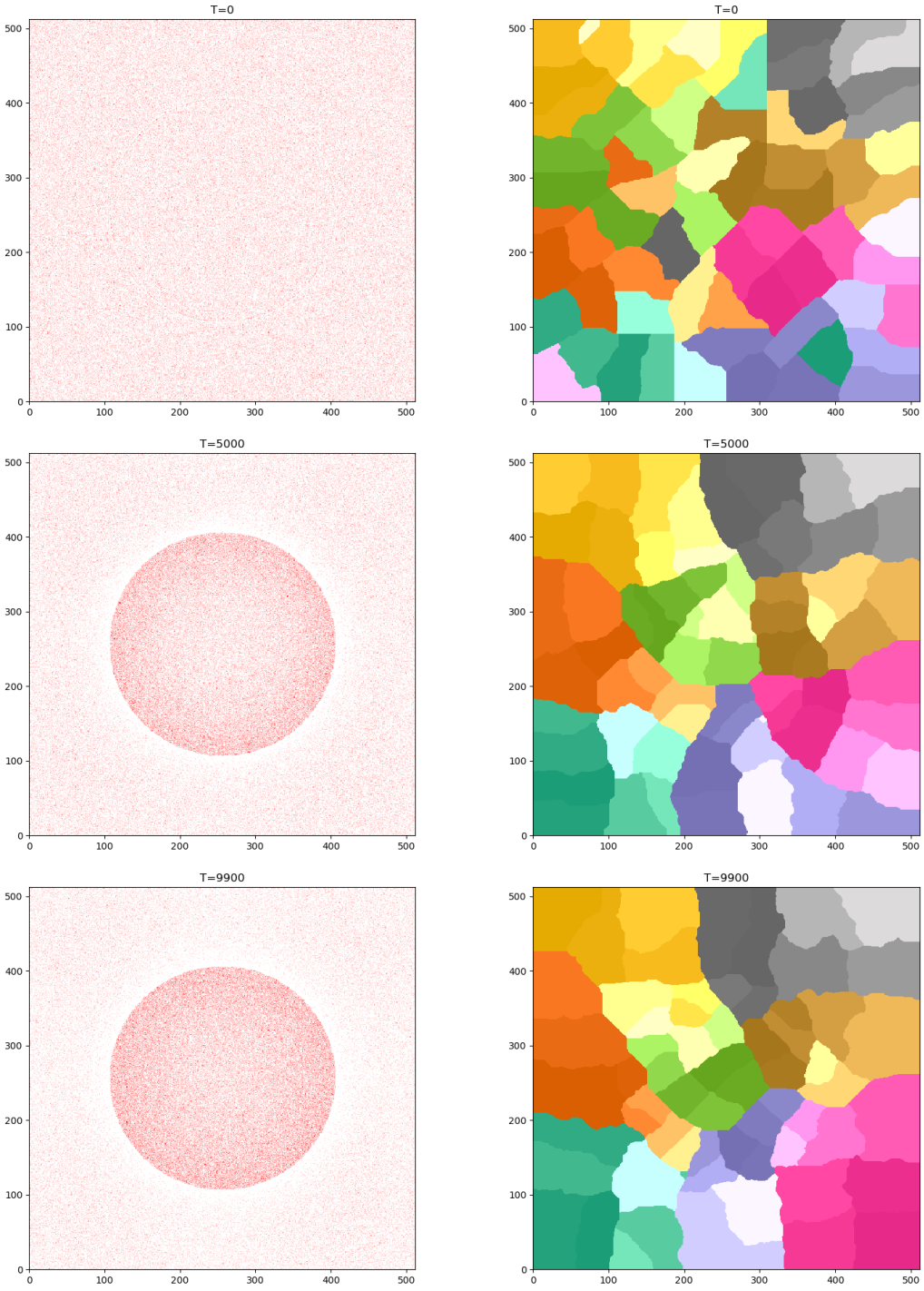
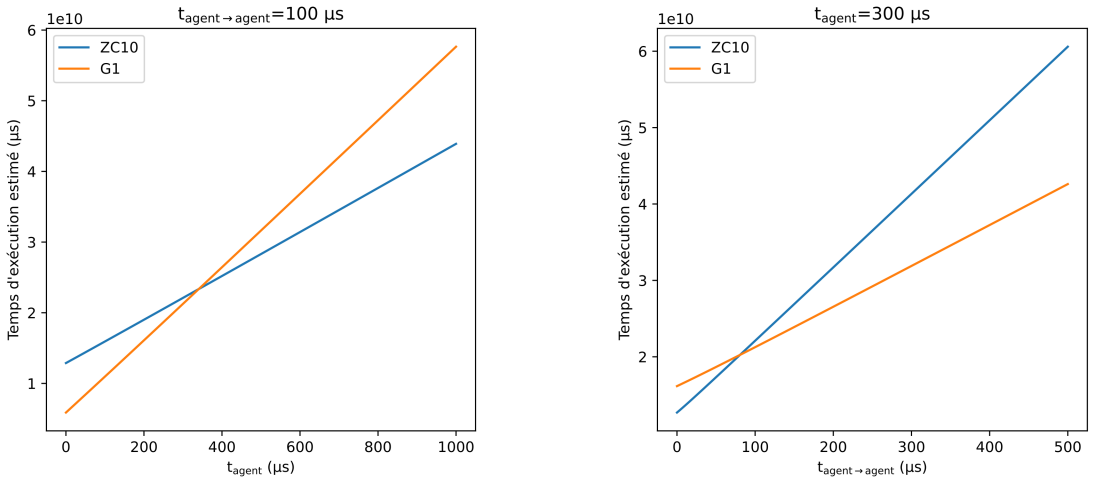


FIGURE 4.12. – Évolution du partitionnement dynamique des cellules réalisé par Zoltan.



(a) En fonction du temps d'exécution des agents.

(b) En fonction du temps de communication des agents.

FIGURE 4.13. – Comparaison des temps d'exécution estimés avec l'équilibrage spatialisé dynamique basé sur Zoltan et l'équilibrage à base de grille statique sur un modèle non uniforme.

cas du déséquilibre observé dans la figure 4.11.

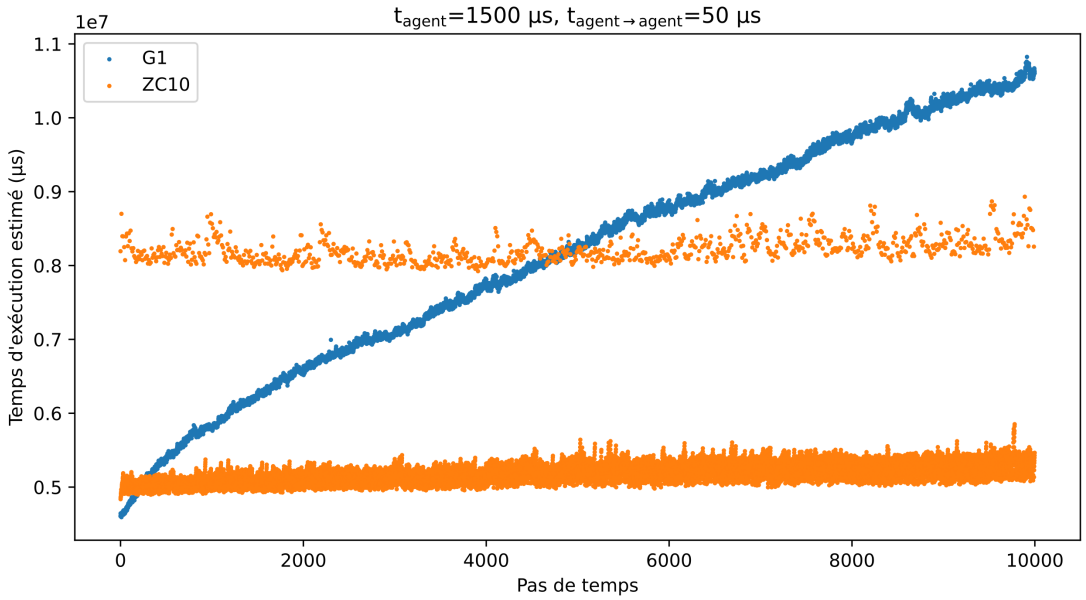
La figure 4.13b présente également une comparaison des temps d'exécution estimés du modèle non uniforme, mais cette fois en fonction du temps de communication entre agents *distants*, après avoir fixé le temps d'exécution des agents à 300 µs. On observe que Zoltan est plus performant pour des temps de communication inférieurs à 90 µs. En effet, Zoltan maintient l'équilibrage de charge entre les processus en toute circonstance, quitte à créer des communications entre les processus qu'il cherche par la suite à minimiser. Or, avec l'algorithme d'équilibrage à base de grille, les agents se concentrent sur quelques processus, minimisant les communications. En cas de communications très coûteuses, les performances globales peuvent s'en trouver améliorées. Nous n'avons pas eu l'occasion d'observer des modèles avec de tels coûts en communication, les temps de communications individuels étant plutôt de l'ordre de la dizaine de microsecondes pour les modèles épidémiologiques par exemple, comme observé au chapitre 5. La question peut cependant se poser dans le cas de modèles nécessitant des échanges de données massifs entre agents.

Pour finir, nous nous intéressons à l'évolution temporelle du partitionnement du modèle non uniforme pour analyser le comportement de Zoltan. Ainsi la figure 4.14 présente les temps d'exécution estimés par pas de temps du modèle non uniforme, dans une configuration de temps d'exécution et de communication choisie pour être favorable à Zoltan, d'après les considérations précédentes. Une première observation à l'échelle globale (figure 4.14a) montre qu'en début de modèle, quand la répartition des agents est encore relativement uniforme, l'utilisation de Zoltan est plus coûteuse que l'équilibrage à base de grille. Cependant, quand la migration des agents vers le centre de

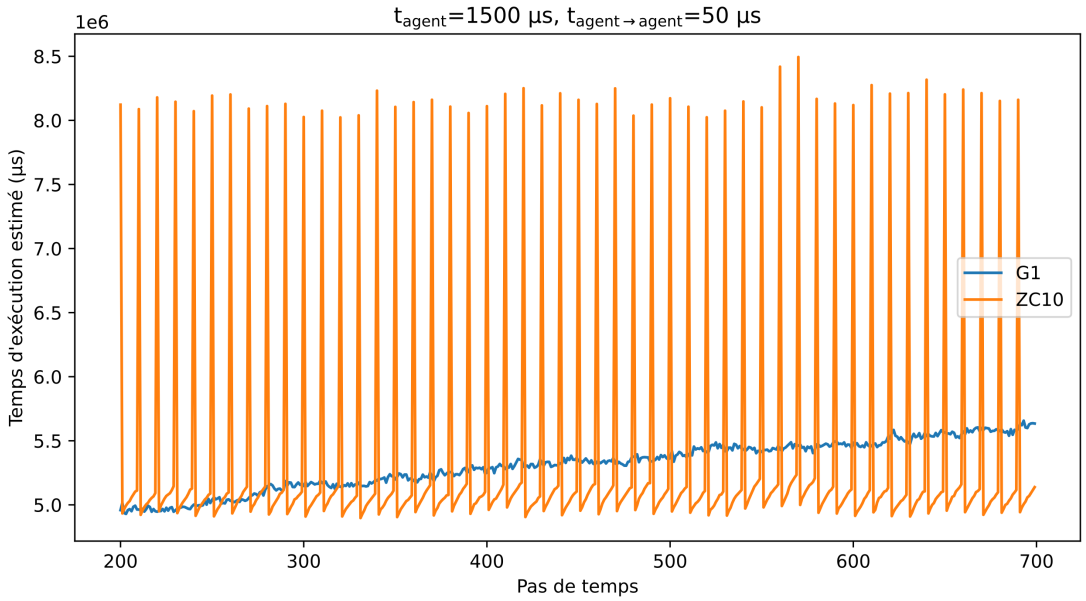
l'environnement entraine un déséquilibre et donc une augmentation des temps d'exécution avec l'équilibrage à base de grille, Zoltan est capable de maintenir un temps d'exécution constant, assurant globalement de meilleures performances. Il est possible d'observer à l'échelle locale (figure 4.14b) l'influence de la période d'application de Zoltan, ici de 10. Chaque pic de temps d'exécution correspond à une exécution de Zoltan. Il est intéressant de noter qu'une application de Zoltan tous les pas de temps aboutirait à un coût global bien supérieur à celui de l'équilibrage à base de grille. Entre les exécutions de Zoltan, les agents se déplacent, augmentant le déséquilibre et donc le temps d'exécution. Avant d'atteindre un déséquilibre critique, l'application de Zoltan permet de systématiquement retomber à un équilibre stable. Il est intéressant de noter que le déséquilibre croît plus rapidement entre les exécutions de Zoltan que sur les mêmes périodes avec l'équilibrage à base de grille. En effet, le déséquilibre avec l'équilibrage à base de grille est purement dû à la migration non uniforme des agents, car les agents sont systématiquement associés au même processus que la cellule dans laquelle ils sont localisés, ce qui permet de garder les agents proches géographiquement sur le même processus et donc de limiter les communications. En revanche, entre deux exécutions de l'équilibrage de charge spatialisé dynamique, aucune migration des agents n'est effectuée. Le déplacement aléatoire des agents entraine alors des coûts de communication supplémentaires. Afin de pallier ce surcoût, il est possible d'imaginer un algorithme d'équilibrage qui applique Zoltan à une période donnée, et qui migre les agents en fonction de leur localisation entre les applications de Zoltan. L'application de Zoltan seulement lorsqu'un seuil de déséquilibre est franchi est également une pratique envisageable. Nous laissons au lecteur la liberté d'imaginer l'implémentation de tels algorithmes grâce à l'interface de synchronisation générique présentée dans ce chapitre.

4.6. Synthèse

L'implémentation de divers algorithmes d'équilibrage de charge dans la plateforme FPMAS prouve la viabilité de l'interface proposée. Celle-ci, grâce à son indépendance par rapport au mécanisme de distribution introduit au chapitre 3, se combine facilement avec lui pour construire une plateforme de simulation distribuée de SMA générique. En effet, par sa simplicité, l'interface permet à la fois de définir des algorithmes applicables à tout SMA et d'autres applicables à des modèles beaucoup plus spécifiques. L'exploration de l'applicabilité de Zoltan à la simulation distribuée de SMA permet bien de rendre compte de cet aspect. Ainsi l'application brute de Zoltan permet de partitionner dynamiquement des modèles à base de graphe pur. En revanche, la prise en compte de l'aspect spatial de certains modèles permet des optimisations significatives de l'utilisation de Zoltan. L'algorithme de partitionnement à base de grille, conçu spécifiquement pour les modèles à base de grille discrète uniformes, est largement plus efficace que Zoltan. L'utilisation de Zoltan peut cependant s'avérer pertinente pour ces modèles lorsque la répartition des agents n'est pas uniforme. D'autres algorithmes, prenant davantage en compte la spatialité des agents dans des modèles non uniformes, ont été évoqués au chapitre 2. Il est difficile de prévoir leurs performances par rapport à Zoltan, mais nos expérimentations



(a) Modèle complet.



(b) Vue locale.

FIGURE 4.14. – Temps d'exécution estimé par pas de temps du modèle non uniforme.

prouvent cependant qu'il est possible d'implémenter ces algorithmes grâce à l'interface d'équilibrage de charge pour les comparer aux autres et s'adapter au mieux à chaque modèle simulé.

Les interactions entre agents ne sont considérées dans ce chapitre qu'en termes de coût de communication. Dans le chapitre suivant, nous nous intéressons à la mise en place des interactions entre agents dans un contexte d'exécution distribuée.

Chapitre 5.

Synchronisation des données

Les travaux présentés aux chapitres 3 et 4 permettent de partitionner efficacement une simulation de SMA, puis de la distribuer tout en assurant la continuité des données. Chaque agent *local* a ainsi la possibilité d'interagir avec des agents exécutés par d'autres processus grâce à un ensemble d'agents *délégués*. Le problème de la *synchronisation des données* consiste alors à définir et mettre en place les modalités d'interactions avec les agents *délégués*, dans un contexte d'exécution par pas de temps. Dans la section 5.1, nous définissons dans un premier temps les interactions entre agents comme des accès en lecture et en écriture à leurs données. Nous identifions ensuite plusieurs règles possibles pour la réalisation de ces interactions, d'où l'introduction de plusieurs *modes de synchronisation*. Nous proposons enfin une interface de synchronisation générique permettant d'implémenter ces modes dans le cadre d'une simulation distribuée, tout en permettant l'implémentation des interactions entre agents sous forme de lectures et d'écritures génériques. L'analyse théorique de ces modes commence avec la section 5.2 par une étude des contraintes imposées aux modèles pour permettre l'utilisation de chaque mode, notamment en termes d'interactions entre agents. Cette analyse se poursuit dans la section 5.3 par une étude des garanties théoriques de reproductibilité associées à chaque mode. Le chapitre se termine par une étude expérimentale des modes de synchronisation implémentés dans FPMAS. Nous étudions d'abord les performances des modes de manière générique grâce au *Méta-Modèle* dans la section 5.4, afin de mesurer les coûts en temps des lectures et des écritures selon chaque mode. Nous étudions ensuite dans la section 5.5 l'impact des modes sur les résultats et la reproductibilité des simulations d'un modèle réel : le modèle *Virus*. Le bilan de ces analyses montre la nécessité pour les plateformes distribuées de simulation de SMA de fournir divers modes de synchronisation afin de s'adapter au mieux aux contraintes des utilisateurs et aux spécificités des modèles.

Table des matières

5.1. Modes de synchronisation

L'étude des plateformes de simulation existantes réalisée au chapitre 2 présente différentes techniques de gestion des interactions entre agents. Les motivations qui influencent les choix des concepteurs de chaque plateforme ne sont cependant pas toujours évidentes, de même que les conséquences sur les performances, la modélisation et les résultats des simulations. C'est pourquoi nous proposons ici la définition d'une interface de synchronisation générique permettant d'implémenter différents modes afin de comparer rigoureusement leurs propriétés théoriques, leurs performances et leurs impacts sur les résultats des simulations.

5.1.1. Lectures et écritures

La description des interactions entre agents par des lectures et des écritures a déjà été abordée au chapitre 2, avec quelques exemples pour des modèles spécifiques.

Nous cherchons ici à formaliser ce concept et à le mettre en lien avec la construction d'une interface de synchronisation de données générique dans le contexte de la simulation distribuée de SMA.

En effet, décrire les interactions par des lectures et des écritures permet de fournir à l'utilisateur final une interface haut niveau pour définir le comportement des agents indépendamment de la distribution, tout en permettant une gestion bas niveau des modalités d'accès aux données.

Dans le cas des SMA, la temporalité des interactions est essentielle. Certains types d'interactions peuvent par ailleurs être interdits, pour des raisons techniques ou pour satisfaire des contraintes de modélisation. Par exemple, Repast HPC permet seulement la lecture sur les agents *délégués* à partir de données importées depuis le pas de temps précédent, et autorise la lecture et l'écriture sur les données du pas de temps actuel pour les agents *locaux*. D-MASON n'autorise que des lectures sur les données *ghost* pour tous les agents, mais autorise l'agent à modifier son propre état. D'autres projets proposent la prise en compte des modifications en fin de pas de temps grâce à des mécanismes de gestion de conflits [116, 103, 107] ou par la modélisation « Influence-Reaction » [56].

Du point de vue de la modélisation d'un SMA, les modalités d'interactions entre agents peuvent être contraintes ou indéfinies [88]. Par exemple, on peut imposer au niveau d'un modèle de nuées d'oiseaux que chaque agent doive lire la position des autres depuis le pas de temps précédent ou depuis le pas de temps en cours ou ne pas spécifier de contrainte. Le passage à la simulation distribuée pourra alors impliquer l'usage de modes de synchronisation spécifiques pour que ces contraintes soient correctement appliquées aux interactions avec les agents *délégués*.

De manière générale, pour respecter le principe de causalité, un agent exécuté au pas de temps T peut lire des données depuis les pas de temps T ou $T-1$, et écrire des données aux pas de temps T ou $T+1$. Une lecture au pas de temps $T-1$ correspond à un accès à une copie *ghost*. Une lecture au pas de temps T signifie que les écritures déjà effectuées au temps T par d'autres agents sont perçues. Une écriture au temps T implique que les agents effectuant des lectures au temps T pourront percevoir les modifications pendant le

Mode de synchronisation	*soi-même* écriture	*local* lecture	*local* écriture	*distant* lecture	*distant* écriture
`GhostMode`	T	T	T	$T-1$	$\times$
`GlobalGhostMode`	$T+1$	$T-1$	$\times$	$T-1$	$\times$
`HardSyncMode`	T	T	T	T	T
`PushGhostMode`	T	T	T	$T-1$	$T+1$
`PushGlobalGhostMode`	$T+1$	$T-1$	$T+1$	$T-1$	$T+1$

TABLE 5.1. – Aspect temporel des interactions pour différents modes de synchronisation.

pas de temps en cours, alors que les écritures au temps $T+1$ seront seulement accessibles au pas de temps suivant. Les écritures peuvent également être interdites dans certains contextes. Pour chaque opération, il est nécessaire de spécifier le type d'agent cible avec lequel un agent interagit :

— *soi-même* : l'agent cible est l'agent lui-même ;
— *local* : l'agent cible est exécuté sur le même processus que l'agent effectuant l'interaction ;
— *distant* : l'agent cible est exécuté sur un autre processus. L'interaction a lieu par l'intermédiaire d'un agent *délégué*.

Grâce à ces concepts, nous définissons formellement plusieurs modes de synchronisation, présentés au tableau 5.1 et déjà publiés à l'issue de la conférence PAAMS 2022 [30].

Le *GhostMode* est équivalent à la synchronisation des données réalisée par Repast HPC. Il consiste à réaliser les lectures distantes depuis une copie *ghost* du système. Les lectures et écritures sur les agents locaux sont réalisées sur place au temps T. Le *GlobalGhostMode*, équivalent à D-MASON, impose en plus l'accès en lecture aux agents locaux depuis une copie *ghost*. L'accès en écriture n'est autorisé que sur l'agent lui-même, mais l'accès *ghost* fait que ces modifications ne peuvent être visibles qu'au pas de temps suivant. Le *HardSyncMode*, développé spécialement dans le cadre de ce projet, permet de réaliser de manière transparente des lectures et des écritures concurrentes au pas de temps T sur tous les agents, y compris sur les agents *distants*. Les variantes *PushGhostMode* et *PushGlobalGhostMode* permettent d'ajouter aux modes correspondants la possibilité de réaliser des écritures en fin de pas temps, par exemple grâce à des mécanismes de gestion de conflits. Ces deux derniers modes n'ont, à ce jour, pas été implémentés mais permettent d'illustrer la flexibilité de l'interface de synchronisation générique.

5.1.2. Interface de synchronisation

La définition d'une interface de synchronisation générique consiste à fournir à l'utilisateur des méthodes abstraites permettant de réaliser des lectures et des écritures sur d'autres agents. Ainsi l'utilisateur final peut implémenter un modèle sans se soucier de l'implémentation réelle des méthodes. Proposer une interface à implémenter permet en outre de simplifier le travail du développeur en décomposant le problème parfois

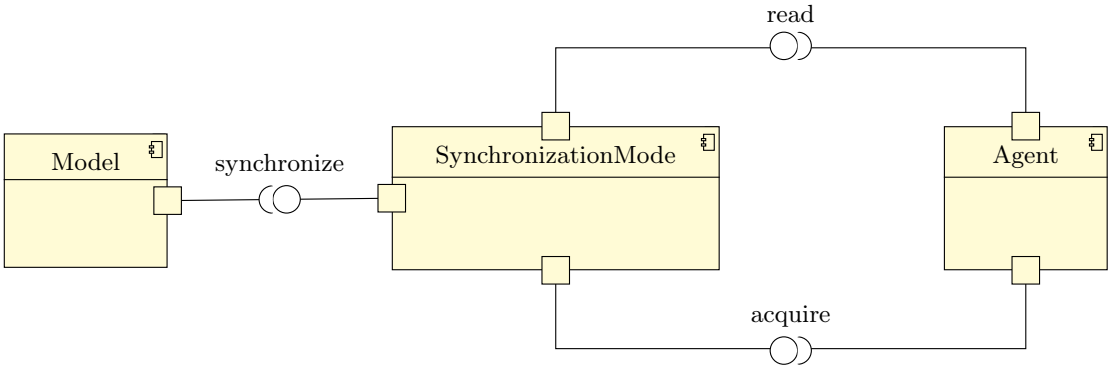


FIGURE 5.1. – Diagramme de composants UML pour l'interface de synchronisation des données.

complexe de la synchronisation des données en sous-problèmes plus simples. Enfin, l'implémentation des modèles étant basée sur des appels aux méthodes de l'interface, il est possible de changer de mode de synchronisation sans altérer le code source des modèles.

La figure 5.1 présente les interactions entre le mode de synchronisation, le modèle et les agents. Le composant `SynchronizationMode` définit un mode de synchronisation qui fournit plusieurs fonctionnalités au modèle et aux agents. La méthode `synchronize` s'applique au modèle à la fin de chaque pas de temps. Plus précisément, chaque processus entre dans la phase de synchronisation après avoir terminé d'exécuter tous les comportements des agents planifiés au pas de temps en cours. Les méthodes `read` et `acquire` permettent aux agents de lire et d'acquérir les données des autres lors de l'exécution de leurs comportements. Contrairement à la lecture, l'acquisition suppose une possible modification des données, ce qui peut impliquer un accès exclusif, selon le mode de synchronisation implémenté. Ces deux opérations ne sont pas considérées comme ponctuelles, mais représentent davantage un travail sur une durée délimitée dans le temps. Le comportement de l'acquisition correspond au paradigme « Read-Modify-Write » [80] : les données acquises sont d'abord lues, toutes les modifications nécessaires sont appliquées, et le résultat est enfin écrit dans les données. La notion d'acquisition fait également référence au fait d'obtenir exclusivement et temporairement les données elles-mêmes pour y appliquer des modifications avant de les rendre. Afin de mettre en place ces mécanismes, il est possible d'implémenter l'interface présentée sur la figure 5.2.

L'interface se base sur des méthodes inspirées des mécanismes de verrouillage issus de la programmation parallèle. Plutôt que de définir des opérations atomiques de lecture et d'écriture sur chaque donnée qu'un agent peut posséder, l'interface permet d'appeler n'importe quelle méthode de l'agent dans des blocs de code protégés par des balises. La seule adaptation des comportements des agents requises pour permettre l'exécution distribuée consiste alors à délimiter les instructions accédant aux données d'autres agents avec les balises suivantes :

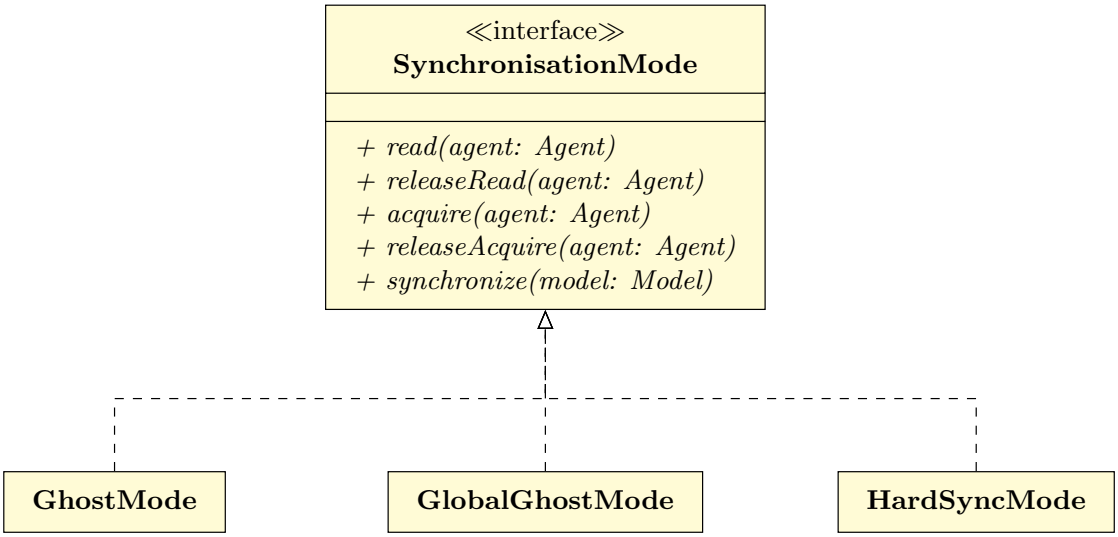


FIGURE 5.2. – Diagramme de classe de l'interface de synchronisation des données et exemples d'implémentation.

`read(Agent)`, `releaseRead(Agent)`

Les données de l'agent spécifié peuvent être lues mais pas modifiées entre les appels aux méthodes `read()` et `releaseRead()`. Ainsi une *lecture* correspond à l'ensemble des opérations exécutées entre les appels à `read()` puis `releaseRead()`. Si l'agent spécifié est *délégué*, la mise à jour interne (ou non) des données qu'il contient dépend du mode de synchronisation implémenté.

`acquire(Agent)`, `releaseAcquire(Agent)`

L'opération `acquire()` donne au processus courant un accès exclusif à l'agent spécifié, de telle sorte que toute modification peut être effectuée sur l'agent jusqu'à l'appel de `releaseAcquire()`. Ainsi une *écriture* correspond à l'ensemble des instructions exécutées entre les appels à `acquire()` puis `releaseAcquire()`. Le comportement réel de ces méthodes dépend de l'implémentation du mode de synchronisation, et ne requiert pas la prise en compte des modifications.

`synchronize(Model)`

Méthode dont le comportement dépend exclusivement du mode de synchronisation implémenté, et dont l'appel est garanti depuis chaque processus suite à l'exécution des agents *locaux* pour le pas de temps en cours. Il est donc possible de mettre en place dans cette méthode des barrières de

synchronisation ou des communications collectives entre les processus pour, par exemple, mettre à jour les données des agents *délégués*.

Afin d'implémenter ces méthodes pour mettre en place les différents modes de synchronisation proposés, nous supposons disponibles, conformément au chapitre 3, les fonctionnalités suivantes :

— des méthodes permettant d'obtenir le voisinage de tout agent *local*, quel que soit le type d'environnement. Ce voisinage peut-être constitué d'agents *locaux* et *délégués*, de sorte qu'il soit possible d'interagir de manière transparente avec des agents exécutés sur d'autres processus.
— des méthodes permettant d'obtenir la localisation de chaque agent *délégué*, c'est-à-dire le processus sur lequel il est exécuté.
— des méthodes permettant d'obtenir la liste complète d'agents *locaux* et *délégués* assignés au processus courant.

Nous présentons dans la suite des algorithmes permettant de mettre en place le `GhostMode`, le `GlobalGhostMode` et le `HardSyncMode` à partir de l'interface de synchronisation des données.

5.1.3. GhostMode

L'algorithme 13 présente un exemple d'implémentation du `GhostMode`. Les méthodes `read()` et `acquire()` retournent simplement l'état de l'agent sur le processus courant, qu'il soit *local* ou *délégué*. Nous supposons que l'exécution de chaque processus est séquentielle, c'est-à-dire que les agents sont exécutés un par un à l'échelle d'un processus. Il n'est donc pas nécessaire de gérer des problèmes de concurrence avec les méthodes `releaseRead()` et `releaseAcquire()`. Ces méthodes pourraient cependant être utilisées pour mettre en place une exécution parallèle des agents à l'échelle de chaque processus. Dans le cas général, la méthode `releaseAcquire()` doit engager les modifications dans l'état actuel de l'agent. Cependant, dans le cas où l'état actuel de l'agent est retourné sous forme de référence ou de pointeur, cette étape est implicite car les modifications sont directement effectuées sur l'état interne de l'agent.

L'essentiel du travail réalisé par le `GhostMode` réside dans la méthode `synchronize()` qui va importer les données à jour des agents *délégués* en fin de pas de temps. L'implémentation proposée ici montre l'usage possible de communications collectives pour réaliser efficacement les échanges de données en initialisant un minimum de communications. Dans le contexte de l'utilisation du protocole MPI, ces échanges sont typiquement réalisés avec la méthode `MPI_AllToAll` ou ses variantes. On remarque que les modifications des données éventuellement acquises ne sont pas prises en compte et effacées par la synchronisation avec les données importées.

5.1.4. GlobalGhostMode

L'algorithme 14 présente un exemple d'implémentation du *GlobalGhostMode*. L'implémentation des méthodes `read()` et `acquire()` est similaire au `GhostMode`, avec le même contexte de gestion de concurrence. La méthode `read()` retourne cependant

Algorithme 13 Implémentation du `GhostMode`.

```
1: algorithme READ(agent)
2:     retourner l'état actuel de l'agent
3: fin algorithme
4: algorithme RELEASEREAD(agent)
5:     -
6: fin algorithme

7: algorithme ACQUIRE(agent)
8:     retourner l'état actuel de l'agent
9: fin algorithme
10: algorithme RELEASEACQUIRE(agent)
11:     mettre à jour l'état actuel avec les modifications effectuées
12: fin algorithme

13: algorithme SYNCHRONIZE(modèle)
14:     requêtes : associe à chaque processus une liste d'ID
15:     pour chaque agent délégué du modèle faire
16:         ajouter l'ID de l'agent aux requêtes de la localisation de l'agent
17:     fin pour
18:     ÉCHANGER AVEC TOUS LES PROCESSUS : requêtes
19:     réponses : associe à chaque processus une liste d'états d'agents
20:     pour chaque processus faire
21:         pour chaque requête du processus faire
22:             ajouter l'état de l'agent aux réponses pour le processus
23:         fin pour
24:     fin pour
25:     ÉCHANGER AVEC TOUS LES PROCESSUS : réponses
26:     pour chaque agent délégué du modèle faire
27:         mettre à jour l'état actuel de l'agent avec les données importées
28:     fin pour
29: fin algorithme
```

une copie *ghost* de l'agent, et non son état actuel. La modification des données n'est pas autorisée dans ce mode, sauf si l'agent réalisant la modification est l'agent lui-même. C'est pourquoi, dans cet exemple d'implémentation, les méthodes `acquire()` et `releaseAcquire()` permettent tout de même les modifications. La responsabilité de ne pas appeler la méthode sur des agents autres que lui-même est donc laissée à l'utilisateur. D'autres implémentations pourraient conduire à une vérification stricte de ces conditions.

La méthode `synchronize()` met d'abord à jour les données des agents *délégués* à la manière du `GhostMode`. Il est ensuite nécessaire d'initialiser, pour tous les agents, *locaux* ou *délégués*, une copie *ghost* de l'agent qui sera retournée par la méthode `read()` jusqu'à la prochaine synchronisation. On remarque que même si l'agent met à jour ses propres données, la copie *ghost* n'est pas altérée et les autres agents lisent toujours son état au pas de temps précédent.

Algorithme 14 Implémentation du `GlobalGhostMode`.

```
1: algorithme READ(agent)
2:     retourner une copie ghost de l'agent
3: fin algorithme
4: algorithme RELEASEREAD(agent)
5:     -
6: fin algorithme

7: algorithme ACQUIRE(agent)
8:     retourner l'état actuel de l'agent
9: fin algorithme
10: algorithme RELEASEACQUIRE(agent)
11:     mettre à jour l'état actuel avec les modifications effectuées
12: fin algorithme

13: algorithme SYNCHRONIZE(modèle)
14:     GHOSTMODE.SYNCHRONIZE(modèle)
15:     pour chaque agent du modèle faire
16:         initialiser la copie ghost de l'agent avec son état actuel
17:     fin pour
18: fin algorithme
```

5.1.5. HardSyncMode

L'implémentation du `HardSyncMode` nécessite la mise en place d'une architecture nettement plus lourde que les modes précédents. Les principaux enjeux sont les suivants :

- Les modifications d'un agent *délégué* doivent être reportées à l'agent *distant* correspondant.
- La modification d'un agent *local* ou *délégué* doit se faire lors d'un accès exclusif, de sorte qu'aucun processus ne puisse lire ou modifier les données de l'agent jusqu'à

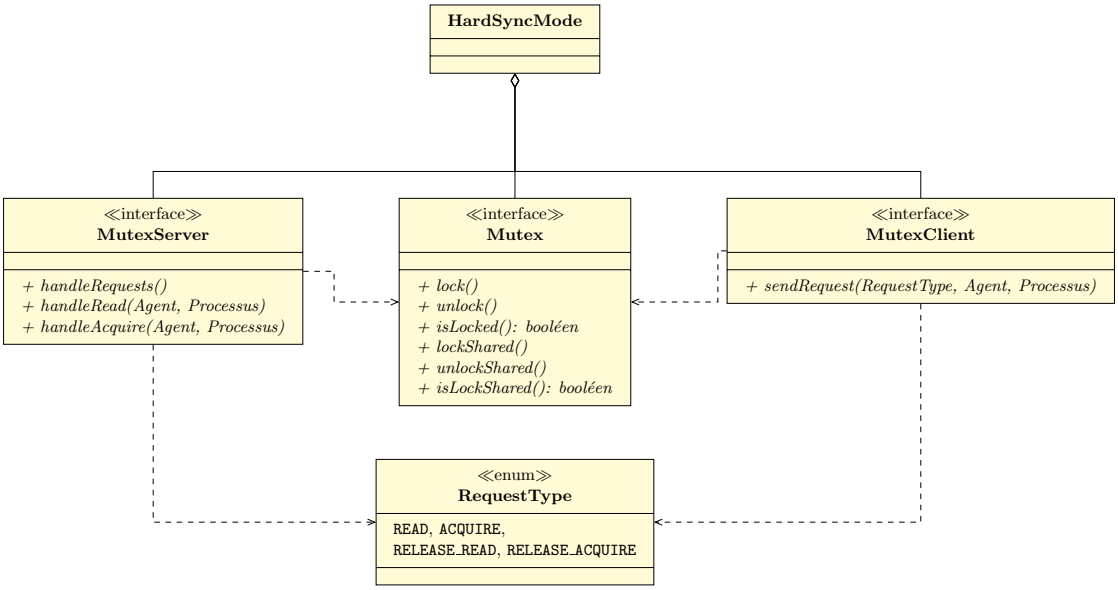


FIGURE 5.3. – Diagramme de classe des composants associés au `HardSyncMode`.

ce que les modifications soient engagées.

— La lecture d'un agent doit retourner des données incluant toutes les modifications effectuées jusqu'à présent sur l'agent, y compris par d'autres processus.

Il est possible de concevoir de nombreuses implémentations du `HardSyncMode` compatibles avec l'interface de synchronisation. Notre proposition se base sur trois composants, introduits dans la figure 5.3 et détaillés dans la suite. Le `Mutex` permet de gérer la concurrence d'accès à chaque agent *local*. Le `MutexClient` permet de réaliser des requêtes pour obtenir l'accès aux agents *délégués*, et le `MutexServer` permet de répondre à ces requêtes avec les données des agents *locaux*.

Mutex

Le `Mutex` inclut des fonctionnalités de verrouillage permettant d'assurer l'accès exclusif à une donnée grâce à la méthode `lock()`. La méthode `lockShared()` permet un accès partagé. Lorsqu'un `lock()` est acquis, il n'est pas possible d'obtenir un autre `lock()` ou un `lockShared()`. Lorsqu'un `lockShared()` est acquis, il est possible d'acquérir d'autres `lockShared()` mais pas de `lock()` exclusif. La fonction de déverrouillage `unlock()` permet de lever un verrouillage exclusif. La fonction `unlockShared()` lève seulement un des verrouillages partagés actuellement actifs. Ainsi la méthode `isLockShared()` renvoie vrai si et seulement si au moins un verrouillage partagé est actif. Si une demande de verrouillage est effectuée sur un `Mutex` indisponible, le processus courant *attend* jusqu'à sa disponibilité. Ainsi les méthodes considérées ici sont dites *bloquantes*. Afin de protéger l'accès des données des agents, une instance de `Mutex` est associée à chaque agent. Les algorithmes présentés dans la suite supposent une exécution des agents en tâche simple. Il est cependant clair que le `Mutex` peut facilement permettre de gérer l'accès cohérent

aux agents dans le cadre d'une exécution multi-tâches des agents.

MutexClient

Le rôle principal du `MutexClient` consiste à transmettre les demandes de lecture, d'acquisition et de fin des ces opérations au `MutexServer` concerné, comme présenté dans l'algorithme 15. L'envoi de la requête (ligne 2) peut se décliner en deux variantes. Si l'agent est *délégué*, la requête doit être transmise à un autre processus. Dans un contexte MPI, cet envoi est typiquement réalisé sous forme de communication point à point grâce à la méthode `MPI_Send` ou à ses variantes. Si l'agent est *local*, la requête peut être transmise directement au `MutexServer` local. Dans tous les cas, les requêtes doivent être mises en attente jusqu'à la disponibilité des données. D'un point de vue algorithmique, le processus courant n'a pas de priorité d'accès à ses propres données par rapport aux autres processus.

Afin de permettre aux autres processus de répondre aux requêtes, une solution directe consiste à recourir à une exécution multi-tâches pour exécuter une instance de `MutexServer` en parallèle de la tâche principale. La tâche associée au `MutexServer` peut répondre aux requêtes grâce à la méthode `handleRequests()`. La tâche principale est quant à elle chargée d'exécuter les agents et peut attendre passivement l'aboutissement des requêtes dans la méthode `sendRequest()`. Cependant, afin de favoriser la traçabilité et le déterminisme de l'exécution de la simulation, nous avons choisi une implémentation basée sur une seule tâche. Le `MutexServer` s'exécute alors ponctuellement au sein de la tâche principale grâce aux appels à la méthode `handleRequests()` (ligne 4), pendant l'attente des réponses aux requêtes du processus courant (ligne 3). L'avancement des processus est ainsi garantie, sans possibilité de blocage. Dans une configuration expérimentale où chaque processus est assigné à un unique cœur de processeur, le fait que chaque cœur n'ait à exécuter qu'une unique tâche permet de s'affranchir des coûts liés à une tâche secondaire passive lorsqu'aucune requête n'est en attente, d'où un potentiel gain en performances. Le fait que chaque processus ne puisse répondre aux requêtes que lorsque lui-même en initie une peut cependant s'avérer limitante si certains processus effectuent moins de requêtes que les autres. Nous n'avons pas été confrontés à ce problème dans la pratique, grâce à l'uniformité des systèmes Multi-Agents simulés et à un équilibrage de qualité.

Algorithme 15 Implémentation du `MutexClient`.

```
1: algorithme sendRequest(type, agent, processus)
2:     envoyer à MutexServer du processus : requête (type, agent)
3:     tant que la requête n'a pas abouti faire
4:         MutexServer.handleRequests( )
5:     fin tant que
6: fin algorithme
```

MutexServer

Le rôle du `MutexServer` consiste à répondre aux requêtes pour les agents *locaux*. Il reçoit à la fois des demandes depuis d'autres processus et depuis le processus local. Il est chargé de l'envoi des données des agents *locaux* aux autres processus et de l'application des verrouillages nécessaires, comme présenté dans les algorithmes 16 et 17. Une instance de `MutexServer` peut gérer tous les agents d'un processus mais les verrouillages sont appliqués individuellement et indépendamment à l'échelle de chaque agent. Il est, par exemple, possible de fournir l'accès exclusif à plusieurs agents *locaux*, sans blocage au niveau du processus. Dans le cas d'une demande de lecture (ligne 4), il suffit de vérifier si un verrouillage exclusif est appliqué. Ainsi l'opération de lecture n'est pas bloquée par d'autres opérations de lecture en cours. Si l'opération est possible (ligne 6), un verrouillage partagé est appliqué et une réponse est envoyée au processus demandeur (ligne 35). Comme dans le cas de l'envoi de la requête, la réponse se décline en deux variantes. Si le processus demandeur est le processus local, il suffit de le notifier de la disponibilité des données pour faire aboutir la requête et déclencher la sortie de la boucle d'attente (ligne 3 de l'algorithme 15). Sinon, il est nécessaire d'envoyer les données de l'agent grâce à une communication point à point, ce qui déclenche également l'aboutissement de la requête au niveau du `MutexClient` correspondant. Dans les deux cas, si l'agent n'est pas disponible, la requête doit être ajoutée à une liste d'attente afin d'être traitée quand le verrouillage sera levé. Le traitement de l'acquisition est similaire (ligne 17), à la seule différence que l'acquisition n'est pas possible tant qu'un verrouillage partagé ou exclusif est appliqué, et qu'un verrouillage exclusif est appliqué quand l'opération est effectuée. L'opération de fin de lecture (ligne 11) consiste d'abord à lever un verrouillage partagé. Si le dernier a été levé, il est garanti que plus aucun accès à l'agent n'est en cours : il est donc possible de traiter la prochaine requête éventuellement en attente [1]. Pour la fin de l'acquisition (ligne 25), un déverrouillage exclusif est d'abord appliqué. Par définition, plus aucun verrouillage n'est alors appliqué : il est donc possible de traiter les requêtes en attente. Il est également nécessaire, en plus des données de la requête, de recevoir les données modifiées de l'agent afin de les inclure à l'agent *local*. Les données à jour sont alors envoyées lors du traitement des prochaines requêtes, conformément à la définition du `HardSyncMode`.

Le processus de traitement des requêtes décrit ici donne la priorité aux lecteurs. En effet, supposons qu'une lecture soit en cours. La prochaine demande d'acquisition est mise en attente. En revanche, la prochaine demande de lecture est exécutée, même si elle arrive après la demande d'acquisition. Il est possible d'imaginer l'implémentation d'autres politiques en modifiant la gestion des requêtes entrantes et l'ordonnancement des requêtes en attente, même si ces pistes n'ont pas été explorées.

Implémentation de l'interface de synchronisation

Les propositions d'implémentation du `MutexClient` et du `MutexServer` permettent de mettre en place le `HardSyncMode` présenté dans l'algorithme 18, pour finalement

1. par construction, il ne peut s'agir que d'une demande d'acquisition dans ce cas

Algorithme 16 Implémentation du `MutexServer` (partie 1).

```
 1: algorithme HANDLEREQUESTS
 2:     RECEVOIR DEPUIS tous les processus : requêtes (type, agent, processus)
 3:     pour chaque requête faire
 4:         si type == READ alors
 5:             si ¬ MUTEX(agent).ISLOCKED( ) alors
 6:                 HANDLEREAD(requête)
 7:             sinon
 8:                 Mettre la requête en attente
 9:             fin si
10:         fin si
11:         si type == RELEASE_READ alors
12:             MUTEX(agent).UNLOCKSHARED( )
13:             si ¬ MUTEX(Agent).ISLOCKSHARED( ) alors
14:                 Traiter les requêtes de l'agent en attente
15:             fin si
16:         fin si
17:         si type == ACQUIRE alors
18:             si ¬ MUTEX(agent).ISLOCKED( ) et
19:                  ¬ MUTEX(agent).ISLOCKSHARED( ) alors
20:                 HANDLEACQUIRE(requête)
21:             sinon
22:                 Mettre la requête en attente
23:             fin si
24:         fin si
25:         si type == RELEASE_ACQUIRE alors
26:             MUTEX(agent).UNLOCK( )
27:             si la requête ne provient pas du processus local alors
28:                 RECEVOIR DEPUIS processus source : nouvel état de l'agent
29:                 Mettre à jour l'état de l'agent local
30:             fin si
31:             Traiter les requêtes de l'agent en attente
32:         fin si
33:     fin pour
34: fin algorithme
```

Algorithme 17 Implémentation du `MutexServer` (partie 2).

35: **algorithme** HANDLEREAD(requête)
36: MUTEX(agent).LOCKSHARED( )
37: ENVOYER À MUTEXCLIENT du processus source : agent
38: **fin algorithme**
39: **algorithme** HANDLEACQUIRE(requête)
40: MUTEX(agent).LOCK( )
41: ENVOYER À MUTEXCLIENT du processus source : agent
42: **fin algorithme**

implémenter l'interface de synchronisation générique. Toutes les opérations passent par une requête effectuée grâce au `MutexClient`. Que la demande d'accès s'applique à un agent *local* ou *délégué*, le processus courant est mis en attente jusqu'à ce que l'agent ait été verrouillé par le `MutexServer` depuis lequel l'agent est *local*. Dans le cas d'un agent *délégué*, les méthodes `read()` et `acquire()` reçoivent les données depuis la localisation de l'agent, une fois sa disponibilité assurée. Dans un contexte MPI, cette opération correspond typiquement à l'utilisation de la méthode `MPI_Recv`. Dans le cas de la méthode `releaseAcquire()`, si l'agent n'est pas *local*, il est nécessaire d'envoyer les modifications de l'agent effectuées par le processus courant au processus d'origine.

D'après le principe de la synchronisation temporelle par pas de temps, lorsqu'un processus a terminé d'exécuter ses agents, il doit attendre que tous les autres aient terminé avant de passer au pas de temps suivant. Or, pour assurer la terminaison des autres processus, il est nécessaire de continuer à répondre à leurs requêtes. D'où la méthode `synchronize()`, qui continue de traiter les requêtes grâce au `MutexServer`. Dans le cas où le `MutexServer` s'exécute dans une tâche parallèle, il suffit d'attendre depuis la tâche principale que les autres processus aient terminé.

La détection de la terminaison est relativement simple dans notre contexte. En effet, le traitement d'une requête ne peut pas réactiver le processus courant, c'est-à-dire réactiver le comportement d'agents déjà exécutés et l'émission d'autres requêtes. L'usage d'algorithmes comme celui de DIJKSTRA, FEIJEN et van GASTEREN [51] est donc disproportionné, le problème correspondant davantage à un simple problème de consensus. Afin de détecter la terminaison, nous organisons les processus selon un anneau orienté. Lorsque le processus 0 entre dans la phase de terminaison (méthode `synchronize()`), il envoie un jeton au processus suivant, qui lui-même le transmet au suivant quand il entre en phase de terminaison, ou instantanément s'il l'est déjà. Lorsque le jeton revient au processus 0, un message indiquant la terminaison est envoyé à tous les processus, qui peuvent alors sortir de la méthode `synchronize()` et passer au pas de temps suivant. Une amélioration proposée par ROUSSET [111] consiste à envoyer le premier jeton au processus en face du processus 0 dans l'anneau, pour ensuite propager un jeton de chaque côté de l'anneau, afin de paralléliser les envois de message de terminaison.

Algorithme 18 Implémentation du `HardSyncMode`.

```
1: algorithme READ(agent)
2:     MutexClient.request(READ, agent)
3:     si l'agent n'est pas local alors
4:         recevoir depuis processus : données de l'agent
5:         mettre à jour l'état actuel de l'agent avec les données reçues
6:     fin si
7:     retourner l'état actuel de l'agent
8: fin algorithme
9: algorithme releaseRead(agent)
10:    MutexClient.request(RELEASE_READ, agent)
11: fin algorithme

12: algorithme ACQUIRE(agent)
13:    MutexClient.request(ACQUIRE, agent)
14:    si l'agent n'est pas local alors
15:        recevoir depuis processus : données de l'agent
16:        mettre à jour l'état actuel de l'agent avec les données reçues
17:    fin si
18:    retourner l'état actuel de l'agent
19: fin algorithme
20: algorithme releaseAcquire(agent)
21:    mettre à jour l'état actuel avec les modifications effectuée
22:    si l'agent n'est pas local alors
23:        envoyer à localisation de l'agent : nouvel état de l'agent
24:    fin si
25:    MutexClient.request(RELEASE_ACQUIRE, agent)
26: fin algorithme

27: algorithme SYNCHRONIZE(modèle)
28:    tant que tous les processus n'ont pas terminé faire
29:        MutexServer.handleRequests( )
30:    fin tant que
31: fin algorithme
```

Synthèse

L'implémentation du `HardSyncMode` montre la possibilité de construire une architecture beaucoup plus complexe que le `GhostMode` ou le `GlobalGhostMode` à partir de l'interface proposée, tout en gardant un haut niveau d'abstraction pour l'utilisateur final grâce aux méthodes `read()` et `acquire()`. Ainsi, l'utilisateur peut librement implémenter le comportement des agents sans se soucier des mécanismes complexes qui peuvent avoir lieu en interne de la plateforme pour par exemple permettre les interactions avec les agents *délégués*. De plus, le fait que l'interface soit commune à tous les modes de synchronisation permet à l'utilisateur de développer des modèles indépendamment du problème de synchronisation des données : le choix du mode de synchronisation peut être réalisé librement par la suite à partir d'un simple paramètre. L'aspect interchangeable des implémentations de l'interface de synchronisation permet ainsi de comparer facilement l'application de différents modes sur un même modèle.

5.1.6. PushGhostMode et PushGlobalGhostMode

Contrairement aux modes précédents, qui eux seuls ont été implémentés dans la plateforme FPMAS, les modes `PushGhostMode` et `PushGlobalGhostMode` ne sont pas détaillés ici. Nous fournissons cependant quelques pistes, afin notamment d'illustrer les possibilités d'implémentation de nouveaux modes offertes par l'interface de synchronisation de données.

En termes de lectures, l'implémentation de ces modes est similaire au `GhostMode` et au `GlobalGhostMode`. Il en est de même pour les écritures sur les agents eux-mêmes. Les écritures sur des agents *locaux* en `PushGhostMode` peuvent se faire directement sur les données locales, comme c'est le cas en `GhostMode`.

La difficulté consiste en la gestion des écritures au pas de temps suivant sur les agents *distants* en `GhostMode` et sur les agents *locaux* et *distants* en `PushGhostMode`, comme défini précédemment dans le tableau 5.1. Tous ces cas peuvent être gérés de manière similaire grâce à un mécanisme d'*agrégation* ou de *gestion de conflits* appelé au moment de la synchronisation. Le problème est le suivant : pour chaque agent, il existe un nombre d'états potentiellement divergents issus des modifications effectuées par différents agents, depuis plusieurs processus. Ces modifications sont stockées dans une liste au cours du pas de temps par les appels à la méthode `releaseAcquire()`. Tous ces états sont d'abord envoyé au processus depuis lequel l'agent est *local*. Il est ensuite nécessaire d'agréger ces résultats depuis ce processus sous forme d'un unique état, qui deviendra l'état de l'agent lu au pas de temps suivant. La politique d'agrégation dépend évidemment du modèle et du type des données des agents, et doit être spécifiée par l'utilisateur. Des exemples de fonctions d'agrégation pour différents types de données sont présentés dans le tableau 5.2.

Le processus d'agrégation peut aussi dépendre :

— de l'état de l'agent à modifier. Par exemple : si l'énergie de l'agent est supérieure à 10, alors choisir l'état 1 ;

— de l'état de l'agent effectuant la demande d'écriture. Par exemple : choisir la modification demandée par l'agent le plus proche géographiquement.

Type de données	Fonctions d'agrégation
Pour tout type	Sélection aléatoire
Numérique	Somme Minimum Maximum ...
Tableau	Agrégation récursive des éléments
Chaîne de caractères Liste Conteneur associatif Ensemble ...	Concaténation
Objet Agent	Agrégation des éléments

TABLE 5.2. – Exemples de fonctions d'agrégation pour plusieurs types de données.

Dans tous les cas, les écritures autorisées par ces modes semblent très spécifiques et liées au modèle à distribuer. De manière générale, les modalités d'interactions permises par les modes de synchronisation proposés sont très diverses. D'où la nécessité de faire une analyse plus poussée des propriétés théoriques des modes de synchronisation.

5.2. Limites d'interactions

Nous débutons l'analyse théorique des modes de synchronisation par les contraintes imposées par chacun d'entre eux aux interactions entre agents, qui vont limiter les modèles possibles à simuler avec chaque mode.

Le `GlobalGhostMode` propose le niveau d'interactions le plus contraint. En effet, l'écriture n'est pas autorisée sur un agent autre que lui-même, et les lectures ne peuvent se faire qu'à partir des données du pas de temps précédent. Le `GlobalGhostMode` est cependant le plus adapté lorsque l'utilisation d'un *ghost* est imposée au niveau du modèle lui-même.

Le `GhostMode` est légèrement moins contraint. En effet, comme souligné par les concepteurs de Repast HPC, les écritures sont autorisées sur tous les agents *locaux* mais perdues sur les agents *délégués*. L'utilisation de cette fonctionnalité ne semble cependant pas pertinente dans un contexte où il est souhaitable de développer des modèles indépendamment de leur distribution. En effet, selon la distribution du modèle, des écritures seront perdues de manière inévitable et imprévisible, au risque de violer les règles du modèle, à moins de prendre en compte l'aspect *local* ou *distant* des agents directement dans les règles du modèle, rendant la modélisation et la simulation dépendantes de la distribution. C'est pourquoi l'utilisation d'écritures sur des agents autres que l'agent lui-

même nous semble à proscrire dans le cadre de l'utilisation du `GhostMode`. Les lectures en `GhostMode` sont moins contraintes qu'avec le `GlobalGhostMode`. En effet, les lectures sont réalisées sur les données du pas de temps actuel, sauf pour les agents *délégués*, lus depuis le pas de temps précédent. Contrairement aux écritures, ce fonctionnement ne peut provoquer de violation des règles du modèle, sauf si bien sûr celui-ci impose explicitement la lecture des données *ghost*. La lecture rend ici visibles les modifications des agents *locaux* ayant déjà exécutés leurs comportements pendant le pas de temps, contrairement au `GlobalGhostMode`. La lecture des agents *délégués* se passe comme si les agents *délégués* étaient tous exécutés après les agents *locaux*. Il est clair que ces lectures peuvent impacter significativement les résultats des modèles par rapport au `GlobalGhostMode`, comme nous le verrons dans la section 5.1 dans le cas d'un modèle épidémiologique. La possibilité de percevoir les modifications des voisins déjà exécutés dans le pas de temps en cours peut cependant mener à des simulations plus réalistes. De tels choix sont laissés à l'appréciation de l'utilisateur, qui peut facilement tester différents modes sans altérer ses modèles grâce à l'interface de synchronisation.

Les modes `PushGhostMode` et `PushGlobalGhostMode`, qui héritent respectivement des propriétés du `GhostMode` et du `GlobalGhostMode` en termes de lecture, permettent un niveau d'interactions un peu plus élevé, en permettant la modification des agents au pas de temps suivant. Leur utilisation est cependant fortement liée au modèle à simuler, le choix des fonctions d'agrégation dépendant des règles du modèle. En d'autres termes, la simulation d'un modèle avec ces modes nécessite une structure de modèle très précise. Cependant, ces contraintes ne semblent pas aller à l'encontre des bonnes pratiques de modélisation indépendantes de la distribution. En effet, la définition de règles incluant uniquement la modification au pas de temps suivant et les fonctions d'agrégation ne fait pas directement intervenir la notion de distribution du modèle et d'agents *locaux* ou *distant*. En outre, le formalisme en question peut exister indépendamment de la simulation distribuée, comme dans le cas de la modélisation « Influence-Reaction ».

Pour finir, le `HardSyncMode` permet le plus haut niveau possible d'interactions, au plus proche de la simulation séquentielle. En effet, il s'agit du seul mode permettant, au cours du pas temps, de lire l'état actuel d'un agent *délégué*, incluant toutes les modifications effectuées pendant le pas de temps par tous les processus. Ce mode permet en outre la réalisation d'écritures distantes et concurrentes par tous les processus sur tous les agents, au cours du pas de temps. Ce mode de synchronisation introduit cependant des possibilités de blocage, où plusieurs processus s'attendent mutuellement sans pouvoir avancer. C'est notamment le cas lorsque des interactions nécessitent l'acquisition simultanée de plusieurs agents. Supposons par exemple un cas avec deux processus et un agent sur chaque processus, où chaque agent cherche à réaliser une acquisition simultanée de lui-même et de l'autre agent. Si l'avancée des processus mène au cas où chaque agent n'a acquis que lui-même, les deux agents restent bloqués indéfiniment dans l'attente d'un verrouillage exclusif de l'autre agent. C'est pourquoi, compte tenu de l'implémentation actuelle du `HardSyncMode` dans FPMAS, le comportement des agents ne doit pas réaliser plus d'une acquisition à la fois. Le respect de cette condition est laissé à la charge de l'utilisateur, introduisant une contrainte indésirable liée à la distribution dans le processus de modélisation. L'inclusion d'algorithmes de prévention des blocages

	`GhostMode`	`GlobalGhostMode`	`HardSyncMode`	`PushGhostMode`	`PushGlobalGhostMode`
Nuée d'oiseaux	×	×	×	-	-
Épidémiologique			×	×	×
Proie-prédateur			×		

TABLE 5.3. – Possibilité de simulation de divers modèles avec les modes de synchronisation proposés.

et d'allocation de ressources pourra faire l'objet des perspectives de développement du `HardSyncMode`, afin de lever les dernières contraintes d'interactions entre agents distants.

La discussion autour des besoins en lectures et écritures pour différents modèles déjà fournie dans la section 2.5 permet de facilement définir les modes utilisables pour simuler chaque modèle, comme présenté dans le tableau 5.3.

Le modèle de nuée d'oiseaux, qui ne nécessite que des lectures, peut-être simulé avec tous les modes permettant des lectures à T ou $T-1$, même si les modes `Push` n'apportent rien au modèle dans ce cas. Un modèle épidémiologique avec infection par écriture sur les agents voisins peut-être simulé avec les modes permettant les écritures à T ou $T+1$, car aucune gestion de la concurrence n'est nécessaire. Le modèle proie-prédateur ne peut-être simulé qu'avec le `HardSyncMode`, si sa spécification nécessite la gestion des écritures concurrentes à T. De manière générale, le `HardSyncMode`, qui reste le plus proche de la simulation séquentielle, permet la simulation de la plus large catégorie de modèles.

Le `HardSyncMode` est cependant sujet à d'autres limitations, notamment en termes de reproductibilité.

5.3. Reproductibilité

La reproductibilité est un critère essentiel de validité dans le domaine de la simulation numérique : on attend généralement de simulations exécutées avec la même configuration qu'elles donnent les mêmes résultats. Or, dans un contexte distribué, les modes de synchronisation exercent une influence déterminante sur la reproductibilité des simulations.

5.3.1. Définition

Dans le cadre de la simulation de SMA, nous pouvons considérer deux types de reproductibilité. La reproductibilité stricte consiste à reproduire exactement et à chaque pas de temps le même comportement des agents entre deux simulations, de sorte que d'un point de vue microscopique le déroulé de la simulation soit le même. La reproductibilité statistique consiste à seulement s'assurer que les résultats sont similaires à l'échelle macroscopique.

Le niveau de reproductibilité requis pour la simulation d'un modèle dépend du contexte de l'étude et des besoins de l'utilisateur. La plateforme MASON accorde, par exemple, une importance particulière à la reproductibilité stricte, alors que la plateforme de simulation à évènements SARL n'impose pas de telle contrainte, compte tenu de l'aspect naturellement stochastique des SMA. Dans le cadre du développement d'une

Niveau	Description
0	Aucune contrainte de reproductibilité.
1	Reproductibilité statistique.
2	Reproductibilité stricte à partitionnement et ordre d'exécution fixe.
3	Reproductibilité stricte à partitionnement fixe, indépendamment de l'ordre d'exécution.
4	Reproductibilité stricte indépendamment du partitionnement.

TABLE 5.4. – Niveaux de reproductibilité dans un contexte de simulation distribuée.

plateforme de simulation générique, il est donc pertinent de s'intéresser à la question de la reproductibilité.

La simulation distribuée introduit en outre davantage de contraintes pour permettre la reproductibilité, notamment en raison de l'avancée stochastique des processus les uns par rapport aux autres. Nous avons cependant établi un lien direct entre le niveau de reproductibilité possible à atteindre et le mode de synchronisation des données utilisé.

Nous commençons d'abord par définir différents niveaux de reproductibilité, présentés dans le tableau 5.4.

Le niveau 0 est atteint lorsqu'aucune forme de reproductibilité n'est observée. Deux simulations lancées avec la même configuration peuvent alors donner des résultats significativement différents.

Le niveau 1, déjà défini, est atteint quand les résultats globaux sont similaires.

Le niveau 2 est atteint lorsque la simulation est strictement reproductible en fixant le partitionnement des agents et leur ordre d'exécution à l'échelle de chaque processus. Dans le cas d'un partitionnement dynamique, nous supposons que la séquence de partitionnement est la même entre deux simulations. À noter que fixer le partitionnement des agents implique de fixer le nombre de processus à utiliser.

Le niveau 3 est atteint lorsque, en fixant le partitionnement, deux simulations sont strictement reproductibles indépendamment de l'ordre d'exécution des agents à l'échelle de chaque processus.

Le niveau 4, le plus élevé, est atteint lorsque deux simulations donnent strictement le même résultat quel que soit le partitionnement du modèle, et quel que soit l'ordre d'exécution des agents à l'échelle de chaque processus. C'est le seul niveau qui peut prétendre à l'obtention de résultats indépendants de la distribution de la simulation, et en particulier indépendants du nombre de processus utilisés.

5.3.2. Niveau de reproductibilité maximal

On observe que chaque niveau de reproductibilité est une condition nécessaire au niveau suivant. Il est donc possible d'assigner un niveau de reproductibilité maximal pour chaque mode de synchronisation.

Les niveaux 2, 3, et 4 impliquent que les résultats des simulations soient indépendants de la vitesse d'avancement des processus. Or les communications effectuées dans le

contexte du `HardSyncMode` dépendent de l'avancée relative des processus par rapport aux autres, qui peut varier d'une exécution à l'autre. Le `HardSyncMode` ne peut donc pas dépasser le niveau 1 de reproductibilité. De manière plus générale, tout mode permettant les lectures ou écritures sur des agents *distants* au temps T implique que les interactions et données lues peuvent dépendre de la vitesse d'avancement relative des processus. Tous ces modes atteignent donc au maximum le niveau 1 de reproductibilité. Nous supposons que pour la plupart des modèles, la reproductibilité statistique est effectivement atteinte avec le `HardSyncMode`. Une telle affirmation ne peut cependant se vérifier qu'expérimentalement, au cas par cas pour chaque modèle.

Les modes permettant les interactions au temps T seulement avec les agents *locaux*, comme le `GhostMode` ou le `PushGhostMode`, permettent d'obtenir des exécutions ne dépendant pas de la vitesse d'avancement des processus, car les données *distantes* sont toujours lues à partir du pas de temps précédent. Cependant, les interactions entre agents *locaux* peuvent dépendre de leur ordre d'exécution à l'échelle de chaque processus. Ces modes peuvent donc atteindre le niveau 2, sans pouvoir le dépasser.

Les modes permettant seulement les lectures au temps $T - 1$, y compris pour les agents *locaux*, permettent de rendre le comportement des agents *locaux* indépendant de leur ordre d'exécution. Les modes `GlobalGhostMode` et `PushGlobalGhostMode` peuvent dont prétendre atteindre le niveau 3 de reproductibilité.

Pour finir, avec les modes `GlobalGhost`, les données lues depuis un agent ne dépendent pas de son état *local* ou *distant*. Ces modes peuvent donc atteindre le niveau 4 de reproductibilité.

5.3.3. Niveau de reproductibilité effectif

Les niveaux de reproductibilité précédents sont les niveaux de reproductibilité maximum atteints par chaque mode, mais l'utilisation des modes ne suffit pas à réellement atteindre le niveau de reproductibilité maximal.

Afin d'étudier la notion de reproductibilité effective, nous définissons l'*état du système* au pas de temps t de la simulation comme l'ensemble de l'état des agents au pas de temps t. Le *mécanisme d'initialisation* du système permet de construire les agents et d'initialiser leurs états au pas de temps 0. Le *mécanisme de transition* permet à l'état des agents et donc du système de passer du pas de temps t à $t + 1$.

Dès lors, une simulation atteint effectivement le niveau de reproductibilité n si et seulement si il est garanti que le mécanisme d'initialisation du système et le mécanisme de transition atteignent le niveau n.

Nous définissons plus précisément l'état d'un agent comme l'ensemble de ses propriétés influençant son comportement ou ses interactions avec les autres, comme par exemple sa position, son champs de perception, ses voisins, sa couleur ou son énergie. En revanche, l'identifiant d'un agent, généré automatiquement et utilisé comme une variable interne nécessaire au fonctionnement du simulateur, ne figure généralement pas dans l'état de l'agent.

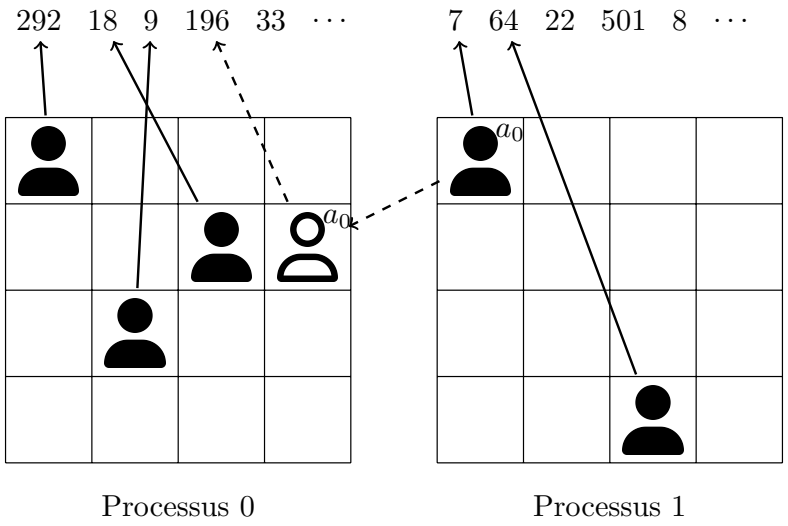


FIGURE 5.4. – Exemple de simulation spatiale distribuée atteignant le niveau 2 de reproductibilité.

Niveau 2

Pour atteindre le niveau 2 de reproductibilité effective pour les modes autorisant les interactions au temps T seulement sur les agents *locaux*, il suffit généralement de s'assurer que le mécanisme de génération de nombres aléatoires atteint le niveau 2, ce qui est notamment le cas avec les générateurs de nombres distribués déterministes introduits à la section 3.5.2. Si l'initialisation du système et le comportement des agents se basent exclusivement sur ce type de générateurs, les mécanismes d'initialisation et de transition, et donc la simulation, atteignent effectivement le niveau 2 de reproductibilité.

Un exemple est présenté sur la figure 5.4. Chaque processus possède sa propre séquence déterministe de nombres aléatoires. On constate, d'une part, que dans ce cas le mécanisme de transition dépend de l'ordre d'exécution des agents, car les nombres aléatoires consommés par chaque agent sur chaque processus dépendent de leur ordre d'exécution. À noter que le problème se pose aussi dans le cas de la simulation séquentielle. D'autre part, le mécanisme de transition dépend aussi de la séquence de partitionnement, car les nombres consommés par les agents dépendent du processus auquel ils sont assignés. Ainsi, quand a_0 migre, il consomme les nombres du processus 0. Ce schéma d'exécution peut dont atteindre la reproductibilité stricte à ordre d'exécution et séquence de partitionnements fixes, mais ne peut la dépasser.

Niveau 3

L'utilisation des modes avec lectures à $T-1$ sur tous les agents ne constitue pas une condition suffisante pour atteindre les niveaux de reproductibilité 3 et 4.

Pour effectivement atteindre le niveau 3, qui consiste à obtenir une simulation reproductible indépendamment de l'ordre d'exécution des agents à l'échelle de chaque

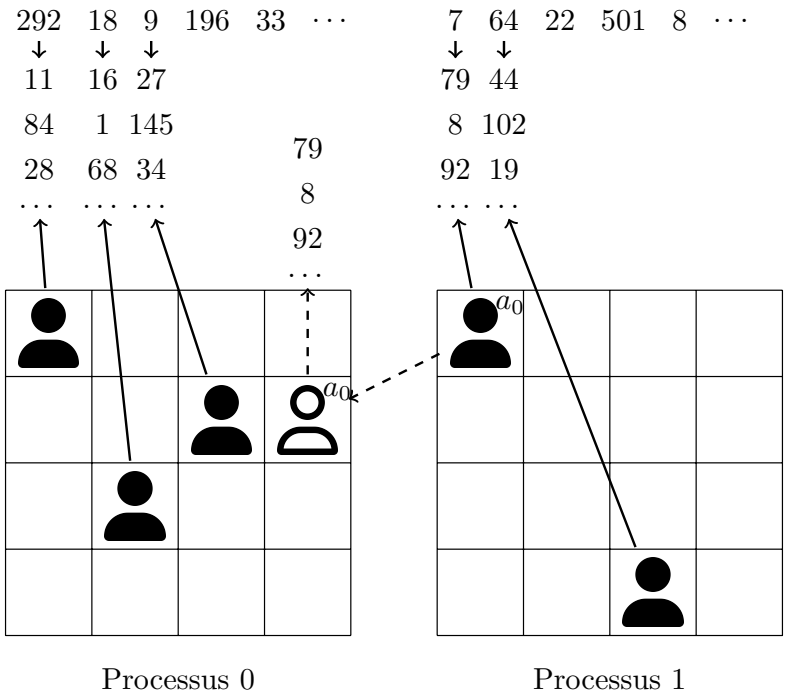


FIGURE 5.5. – Exemple de simulation spatiale distribuée atteignant le niveau 3 de reproductibilité.

processus mais avec une séquence de partitionnements fixée, il est nécessaire que les mécanismes d'initialisation et de transition atteignent également le niveau 3. Les lectures à $T-1$ sur les agents locaux garantissent en partie des interactions entre agents indépendantes de leur ordre d'exécution. Cependant, comme présenté dans l'exemple précédent, utiliser des instances de générateurs à l'échelle du processus ne permet pas d'atteindre le niveau 3.

Afin d'obtenir un mécanisme de transition avec un niveau 3 de reproductibilité, nous proposons le schéma d'exécution illustré dans la figure 5.5.

La méthode consiste à embarquer un générateur de nombres aléatoires avec chaque agent, de sorte que les nombres que chacun consomme ne dépendent plus de leur ordre d'exécution par rapport aux autres.

Pour que le mécanisme d'initialisation atteigne le niveau 3, l'état des agents et les graines de leurs générateurs doivent être définis de manière déterministe et indépendante de l'ordre d'exécution des agents. De plus, compte tenu du grand nombre de générateurs à initialiser, les graines des générateurs doivent être convenablement distribuées afin de ne pas introduire de biais dans la simulation. Le partitionnement étant fixe dans ce cas, ces objectifs peuvent être atteints en générant des graines à partir d'une instance de générateur distribué déterministe au niveau du processus.

La combinaison de ces deux mécanismes permet d'atteindre la reproductibilité de niveau 3 avec les modes n'autorisant que les interactions à $T-1$. On remarque d'autre

part, toujours sur la figure 5.5, que les nombres consommés par chaque agent ne dépendent plus de leur migration. Ainsi, contrairement au cas de la figure 5.4, quand l'agent a_0 migre sur le processus 0, il continue de consommer des nombres depuis le générateur qui lui a été assigné, indépendamment du processus auquel il est assigné. Dans la pratique, la migration du générateur avec l'état de l'agent ne pose pas de problème, car l'état des générateurs déterministes comme le générateur congruentiel linéaire ou le Mersenne Twister sont facilement sérialisables. Le mécanisme de transition est donc indépendant de la séquence de partitionnement, et donc de niveau 4. Cependant, le mécanisme d'initialisation n'atteint pas le niveau 4 : en effet, l'initialisation du générateur de nombres aléatoires dépend ici du processus sur lequel est initialisé a_0. Le fait que la position initiale des agents dans le modèle soit indépendante du nombre de processus ou du partitionnement initial n'est pas non plus garanti dans ce cas.

Niveau 4

Nous détaillons dans cette section une méthode générique pour obtenir un mécanisme d'initialisation de niveau 4, en l'illustrant par la figure 5.6.

La solution consiste à considérer une séquence d'agents de taille N, avec un état indéfini, ou vide. Dans le cadre de l'exécution distribuée, initialiser les agents selon un partitionnement arbitraire consiste alors simplement à initialiser le nombre d'agents nécessaires sur chaque processus. L'instanciation des agents s'effectue donc de manière distribuée, mais indépendamment du partitionnement ou du nombre de processus car à ce stade les états des agents sont vides et donc nécessairement tous égaux. Un indice i (séquence I sur la figure 5.6) est alors associé à chaque agent de manière arbitraire.

La séquence de coordonnées aléatoires POS de taille N est ensuite initialisée grâce à un générateur de nombres aléatoires séquentiel, initialisé indépendamment du partitionnement et du nombre de processus. La séquence générée est donc commune à tous les processus. Les coordonnées à l'indice i sont alors associées à l'agent i. La séquence de graines aléatoires RNG de taille N est ensuite initialisée de la même façon, puis un générateur est associé à chaque agent de sorte que la graine à l'indice i soit utilisée par l'agent i : la même graine est ainsi toujours associée aux mêmes coordonnées. Les éléments de chaque tableau ne dépendant pas du nombre de processus ou de l'assignation des indices, l'état initial des agents, et donc le mécanisme d'initialisation, atteignent alors le niveau 4 de reproductibilité.

Ce mécanisme d'*initialisation en séquence*, qui consiste à attribuer une valeur à chaque agent, est généralisable à tout modèle en initialisant des tableaux de valeurs de taille N pour chaque propriété des agents avec un niveau 4 de reproductibilité.

Nous définissons enfin un mécanisme reproductible d'*initialisation d'un échantillon*, qui permet d'associer une valeur v seulement à un sous-ensemble d'agents. Ce cas correspond par exemple à l'infection d'un nombre n d'agents au début de la simulation d'un modèle épidémiologique. Dans ce cas, un échantillon de n indices est créé de manière reproductible sur tous les processus grâce à un générateur de nombres aléatoires séquentiel, puis la valeur v est associée aux agents correspondant aux indices sélectionnés.

Pour les deux mécanismes d'initialisation, il est possible de considérer des fonctions

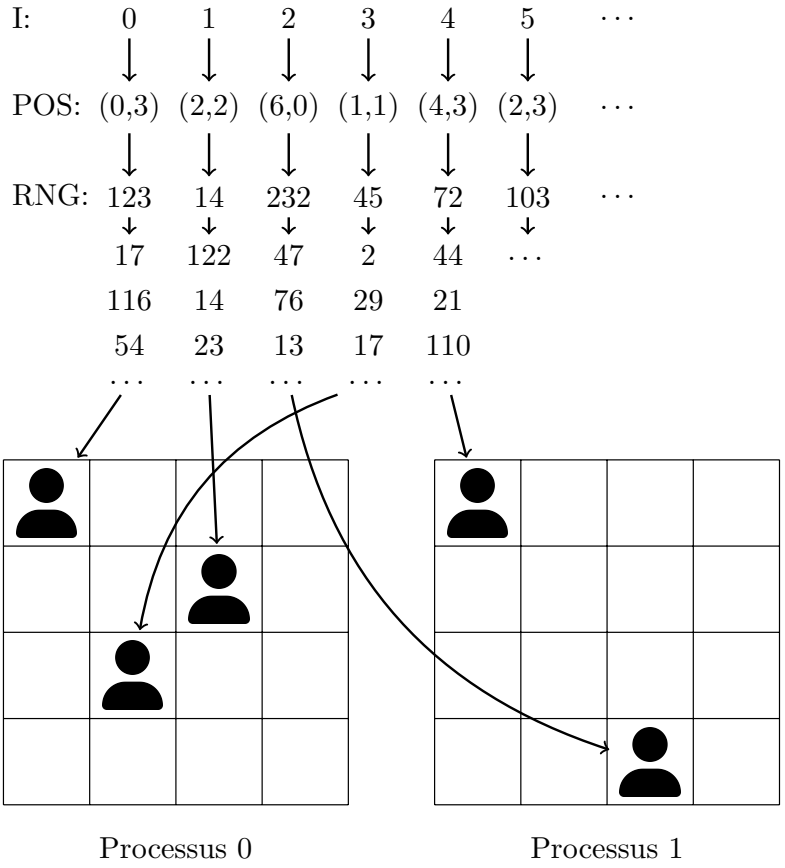


FIGURE 5.6. – Exemple de simulation spatiale distribuée atteignant le niveau 4 de reproductibilité.

d'initialisation plutôt que des valeurs. Leurs appels sont alors naturellement distribués car réalisés seulement sur les processus associés aux agents concernés.

La combinaison de ces mécanismes d'initialisation avec les modes de synchronisation ne permettant que les lectures à $T-1$ permettent d'obtenir une reproductibilité stricte de niveau 4, dans laquelle le déroulement microscopique des modèles et donc leurs résultats globaux ne dépendent pas du nombre de processus utilisés.

Dans l'exemple précédent, la localisation des agents est décrite comme de simples coordonnées. Dans le cas d'un environnement à base de graphe, la localisation est donnée par un nœud du graphe. Il est alors nécessaire, pour initialiser la localisation des agents de manière reproductible, d'une part de générer l'environnement indépendamment du nombre de processus, puis de générer un tableau de nœuds de taille N indépendamment du nombre de processus.

Actuellement, l'initialisation d'agents avec une reproductibilité de niveau 4 n'est assurée que pour les environnements à base de grille discrète dans FPMAS. La possibilité d'exécuter ce type de modèles spatiaux avec une reproductibilité stricte assurée quel que soit le nombre de processus est déjà une contribution par rapport à notre étude des plateformes existantes.

Nous terminons ici la caractérisation théorique des modes de synchronisation, pour maintenant les aborder d'un point de vue expérimental.

5.4. Performances

Les modes précédemment définis se basent sur des stratégies de communication très diverses, allant de la communication point à point avec gestion stricte de la concurrence du `HardSyncMode` aux communications collectives sans gestion des écritures distantes du `GhostMode`. L'objectif de nos expérimentations consiste donc dans un premier temps à évaluer, de la manière la plus générique possible, le coût en temps de calcul des différents modes de synchronisation.

5.4.1. Modèle test

Afin d'évaluer les coûts de chaque mode de la manière la plus général possible, nous utilisons d'abord une méthode basée sur le *Méta-modèle* introduit au chapitre 4. Nous disposons en particulier des mêmes environnements et types de modèles déjà introduits.

Pour tester les performances en lectures et en écritures des modes, nous définissons un modèle à base de graphe pur de type Small-World avec $N = 1\,000\,000$, $K = 8$ et $p = 0{,}1$. Un tel graphe, avec son haut niveau d'agrégation et sa faible longueur de chemin caractéristique, permet à la fois de générer de nombreuses interactions entre agents *locaux* mais aussi des interactions avec des agents *distants* sur tous les processus, ce qui permet de tester les performances des modes de synchronisation dans différentes situations. Le cas des écritures concurrentes est par exemple digne d'intérêt car chaque agent *local* fait régulièrement l'objet d'écritures depuis des agents *locaux* et *distants* à chaque pas de temps.

Comportement	Description
`READ_ONE`	Lecture d'un agent sélectionné au hasard parmi les voisins.
`READ_ALL`	Lecture de tous les voisins.
`WRITE_ONE`	Écriture d'un agent sélectionné au hasard parmi les voisins.
`WRITE_ALL`	Écriture de tous les voisins.

TABLE 5.5. – Type de comportements en lectures et en écritures.

Le graphe d'interactions est partitionné en début de simulation grâce à Zoltan. Les expériences du chapitre 4 ont déjà prouvé la capacité de Zoltan à équilibrer la charge et les communications entre les processus dans le cas d'un graphe Small-World. Nous utilisons pour ces expériences un taux de déséquilibre de 1,02, pour éviter que des processus ne soient associés à aucun nœud y compris lors de l'utilisation de 64 processus. Nous forçons alors Zoltan à maintenir des communications entre tous les processus, afin d'éviter les biais.

Une fois le partitionnement effectué, les agents exécutent un comportement qui consiste à réaliser des lectures et des écritures, pendant 1000 pas de temps. Afin d'obtenir des résultats de performances génériques, nous enrichissons le *Méta-modèle* avec la définition de plusieurs types de comportements, présentés dans le tableau 5.5. Il est possible de combiner deux types de comportement différents. Par exemple, le comportement `READ_ONE_WRITE_ALL` signifie que chaque agent effectue une lecture sur un voisin choisi au hasard, puis effectue une écriture sur tous ses voisins.

Le volume des données des agents est défini comme un paramètre du *Méta-modèle*, afin d'observer l'influence des volumes de communication.

L'étude de performances suivante permet d'évaluer deux caractéristiques des modes de synchronisation :

1. Les performances globales des modes en lectures et en écritures dans l'environnement considéré selon le nombre de processus.
2. Le temps individuel d'une lecture ou d'une écriture *locale* ou *distante* avec chaque mode.

Pour ce faire, nous mesurons pour chaque expérience, sur chaque processus et à chaque pas de temps, les données suivantes :

— $N_{l,l}$: Nombre de lectures entre deux agents *locaux*
— $t_{l,l}$: Temps total passé à faire des lectures *locales*
— $N_{l,d}$: Nombre de lectures entre un agent *local* et un agent *distant*
— $t_{l,d}$: Temps total passé à faire des lectures *distantes*
— $N_{e,l}$: Nombre d'écritures entre deux agents *locaux*
— $t_{e,l}$: Temps total passé à faire des écritures *locales*
— $N_{e,d}$: Nombre d'écritures entre un agent *local* et un agent *distant*
— $t_{e,d}$: Temps total passé à faire des écritures *distantes*
— t_s : Temps de synchronisation

Les expériences ont été menées avec FPMAS 1.6 [2] et la version 1.1 de l'implémentation

du *Meta-Modèle* [3]. L'intégralité des données brutes sont accessibles librement et de manière pérenne grâce à la plateforme dat@UBFC [28].

5.4.2. Lectures

Nous cherchons dans un premier temps à analyser le coût des lectures selon les modes de synchronisation. La figure 5.7 présente les temps observés dans le contexte d'un modèle où les agents exécutent un comportement de type `READ_ALL`. Les expérimentations ont été menées avec les trois modes implémentés dans FPMAS : `GhostMode` (G), `GlobalGhostMode` (GG) et `HardSyncMode` (HS).

Pour chaque pas de temps, les valeurs mesurées sont sélectionnées à partir du processus maximisant la quantité suivante :

$$t_{l,l} + t_{l,d} + t_{e,l} + t_{e,d} + t_s \tag{5.1}$$

D'autres méthodes sont envisageables, comme la moyenne des valeurs obtenues sur chaque processus. La sélection du maximum permet cependant d'obtenir une borne maximale du coût des lectures, et de considérer un ensemble de valeurs ayant réellement été mesurées sur un processus. Dans la pratique, le taux de déséquilibre de 2% garantit cependant une faible variation des valeurs entre les processus, quelle que soit la méthode de sélection des valeurs. Nous réalisons ensuite la somme des valeurs sélectionnées à chaque pas de temps, par type de valeur.

Nous faisons ensuite la moyenne des temps totaux obtenus sur 10 itérations de chaque expérience réalisées avec des graines aléatoires différentes. La taille des données des agents est fixée à 16 octets, ce qui correspond à un volume relativement faible.

La hauteur totale des barres ne doit pas être perçue comme un temps d'exécution total d'un modèle réel. En effet, les agents effectuent des lectures mais n'ont pas de comportement réel basé sur les données lues. En outre, nous ne mesurons que les temps passés dans les méthodes de synchronisation, de lectures et d'écritures, afin de s'affranchir au maximum des biais liés à des détails d'implémentation de la plateforme qui ne sont pas liés aux modes de synchronisation. La somme pour chaque pas de temps des valeurs sélectionnées par le maximum de la quantité présentée dans l'équation 5.1 ne constitue pas non plus une estimation exacte du temps d'exécution réel du modèle.

On observe tout d'abord, à partir de deux processus, une bonne diminution des temps d'interactions en fonction du nombre de processus quel que soit le mode. Ce résultat s'explique principalement par la répartition de la charge et des communications permises par Zoltan, mais prouve également que l'augmentation du nombre de processus n'engendre pas une explosion des temps de communication réels. Dans certains cas, l'exécution avec un seul processus, qui n'implique aucune communication, est plus efficace que l'exécution avec plusieurs processus. En effet, nous considérons ici que le comportement des agents se résume à des interactions, sans considérer de travail local. Le surcoût dû aux communications compense donc rapidement le gain en temps de calcul induit par la distribution de la charge.

Le temps de synchronisation du `GlobalGhostMode` avec un processus correspond essentiellement au temps de mise à jour des copies *ghost* des agents locaux. De manière

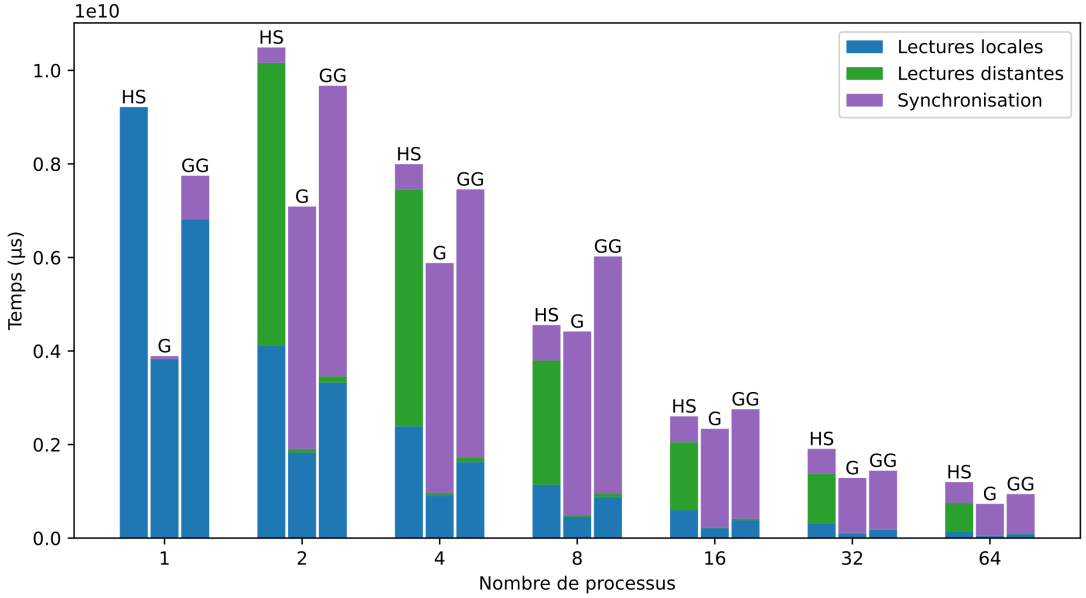
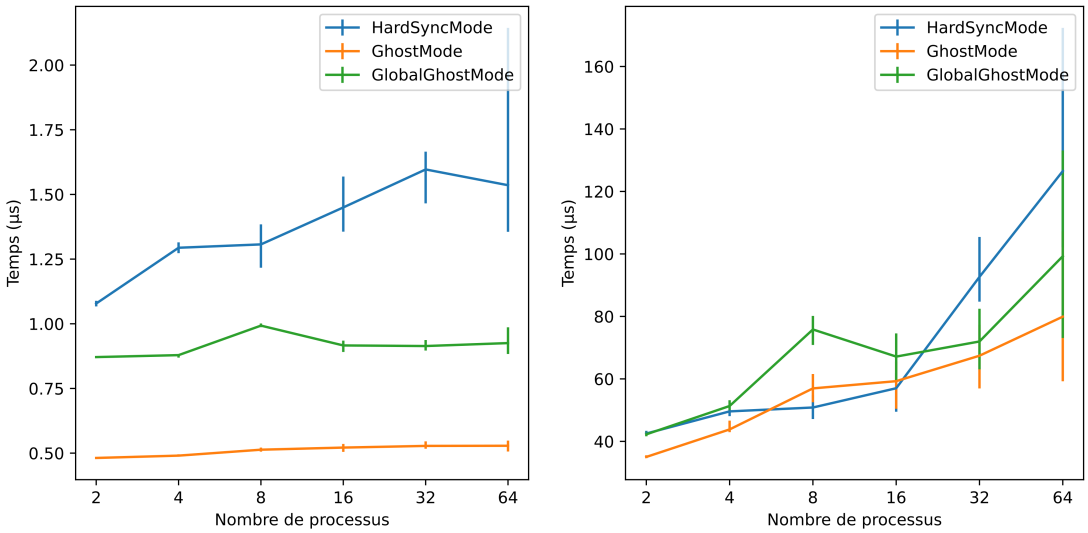


FIGURE 5.7. – Décomposition temporelle des différentes opérations de synchronisation pour un modèle `READ_ALL` sur 1000 pas de temps.

générale, les différences de temps des lectures locales entre les différents modes sont le fruit de détails d'implémentation de FPMAS, tels que la gestion des copies des agents ou de la concurrence. Ce temps est systématiquement le plus faible pour le `GhostMode`, car, dans ce cas, notre implémentation des lectures consiste à trivialement renvoyer une référence vers l'agent.

Le temps des lectures distantes est négligeable pour les modes `GhostMode` et `GlobalGhostMode` par rapport au `HardSyncMode`. En effet, dans ces deux modes, la lecture d'agents *délégués* est similaire à la lecture des agents *locaux*. En revanche, on observe que les communications point à point nécessaires au `HardSyncMode` représentent un temps significatif. Le temps de synchronisation associé à ce mode, qui consiste simplement à attendre la fin de l'exécution du pas de temps par les autres processus en répondant aux dernières requêtes, est bien plus faible que dans le cas des autres modes. En effet, l'essentiel de la charge de travail associée aux lectures distantes pour les modes `GhostMode` et `GlobalGhostMode` se concentre sur l'étape de synchronisation, qui consiste à importer les données à jour de tous les agents *délégués*.

Nous poursuivons notre analyse avec une estimation du coût individuel nécessaire à la réalisation d'une lecture locale ou distante pour chaque mode. L'analyse précédente montre que selon le mode, le coût d'une lecture peut se retrouver dans les étapes de lecture en elle-même ou de synchronisation selon différentes proportions. Afin de pouvoir comparer le coût individuel des lectures selon les modes, nous proposons d'abord d'évaluer



(a) Lectures locales.

(b) Lectures distantes.

FIGURE 5.8. – Temps de lectures individuelles pour un modèle `READ_ALL`.

le coût d'une lecture locale grâce à la formule suivante :

$$t'_{l,l} = \frac{t_{l,l}}{N_{l,l}} \tag{5.2}$$

Nous considérons donc ici que les lectures locales n'ont aucune influence sur le temps de synchronisation. En revanche, nous évaluons le temps d'une lecture distante grâce à la formule suivante :

$$t'_{l,d} = \frac{t_{l,d} + t_s}{N_{l,d}} \tag{5.3}$$

Nous répartissons ainsi le coût des lectures distantes entre le temps passé à faire l'opération de lecture elle-même et le temps de synchronisation.

Les coûts individuels des lectures sont présentés sur la figure 5.8. Les valeurs sont obtenues en appliquant les formules 5.2 et 5.3 à la somme des valeurs sélectionnées à chaque pas de temps selon le même procédé que pour la figure 5.7. La valeur finale est obtenue par la moyenne de 10 expériences, et les valeurs minimales et maximales observées sont représentées grâce aux barres d'erreurs.

On observe d'abord que ces résultats justifient les ordres de grandeurs de coût des lectures distantes utilisés au chapitre 4, ainsi que l'aspect négligeable des temps de lectures locales par rapport aux lectures distantes.

On constate ensuite que la visualisation des coûts de lectures locales confirment le surcoût du `GlobalGhostMode` par rapport au `GhostMode`, qui correspond à la gestion des copies *ghost*, ainsi que le surcoût du `HardSyncMode` dû aux mécanismes de gestion de la concurrence. Le temps individuel de lecture locale est relativement constant en fonction du nombre de processus, malgré une légère augmentation pour le `HardSyncMode`, pour

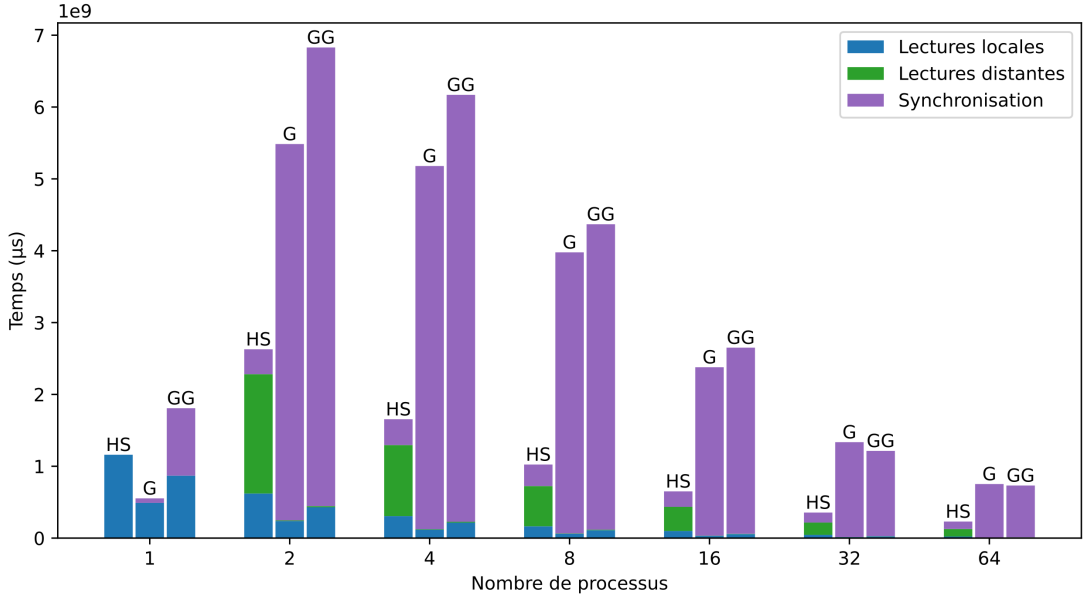


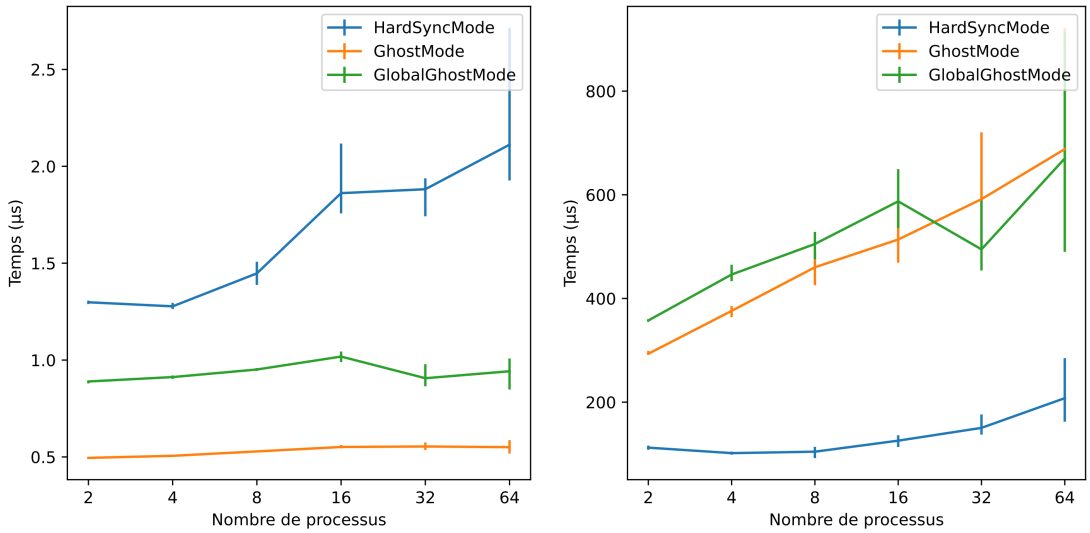
FIGURE 5.9. – Décomposition temporelle des différentes opérations de synchronisation pour un modèle `READ_ONE` sur 1000 pas de temps.

laquelle nous n'avons actuellement pas d'explication précise. Une cause probable pourrait être liée à une trop forte occupation de la mémoire par rapport aux autres modes, ce qui ralentit les accès à la mémoire dans le nœud de calcul, constitué de 16 cœurs. Cette explication est confortée par la réduction de la hausse au delà de 16 processus, encore plus marquée dans les cas `READ_ONE` (figure 5.10) et `WRITE_ALL` (figure 5.12) présentés dans la suite. D'autres causes possibles incluent des incertitudes sur la méthode ou d'autres causes liées à l'implémentation de FPMAS.

L'augmentation du temps de lectures distantes en fonction de tous les modes est nettement plus nette. Nous supposons cette fois que cette augmentation est due à la hausse de complexité des communications avec la hausse du nombre de processus, qu'elles soient collectives (`GhostMode` et `GlobalGhostMode`) ou point à point (`HardSyncMode`). En effet, il est plus simple pour un processus de communiquer avec trois autres processus qu'avec 63 autres, or la structure de graphe Small-World implique des communications avec tous les processus disponibles.

Nous terminons enfin cette analyse avec les résultats d'un modèle avec des comportements de type `READ_ONE`, présentés dans les figures 5.9 et 5.10.

La plupart des considérations précédentes sont encore valides dans ce cas, à l'exception de l'efficacité du `HardSyncMode` par rapport aux autres. En effet, ce mode n'importe les données des agents qu'en cas de lecture effective, contrairement au `GhostMode` et au `GlobalGhostMode` qui importent systématiquement toutes les données des agents *délégués*, même si ceux-ci ne sont pas lus. Les coûts individuels de lectures avec ces derniers modes s'en trouvent grandement augmentés par rapport au cas du modèle



(a) Lectures locales.

(b) Lectures distantes.

FIGURE 5.10. – Temps de lectures individuels pour un modèle `READ_ONE`.

`READ_ALL`, car le nombre de lectures diminue largement mais le temps de synchronisation reste identique.

Le `HardSyncMode`, qui réalise des lectures au temps T avec des communications point à point au cours du pas de temps, n'est donc pas systématiquement plus coûteux que le `GhostMode` et le `GlobalGhostMode`, qui réalisent des lectures au temps $T-1$ avec des communications collectives en fin de pas de temps.

5.4.3. Écritures

Les performances des écritures ne sont évaluées que dans le cas du `HardSyncMode`. En effet, c'est le seul mode parmi ceux implémentés qui permet la réalisation d'écritures distantes. Dans ce contexte, les performances d'un modèle `WRITE_ALL` sont présentées dans la figure 5.11.

Contrairement aux lectures, qui peuvent avoir lieu simultanément, les écritures requièrent un accès exclusif à chaque agent. À noter cependant que malgré la hausse des accès concurrents due à l'augmentation du nombre de processus, le `HardSyncMode` reste efficace jusqu'à 64 processus. Ces résultats sont confirmés par le coût des écritures individuelles présentés dans la figure 5.12. On observe même que ces coûts sont du même ordre de grandeur que le coût des lectures individuelles calculé pour le modèle `READ_ALL` (figure 5.8).

En effet, la réalisation des lectures et des écritures requièrent chacune le même nombre de messages pour demander la ressource, l'importer, et libérer l'opération, à la seule différence que lors d'une écriture il est nécessaire de renvoyer les données à jour. Or le volume de données associées aux agents dans cette expérience est relativement faible

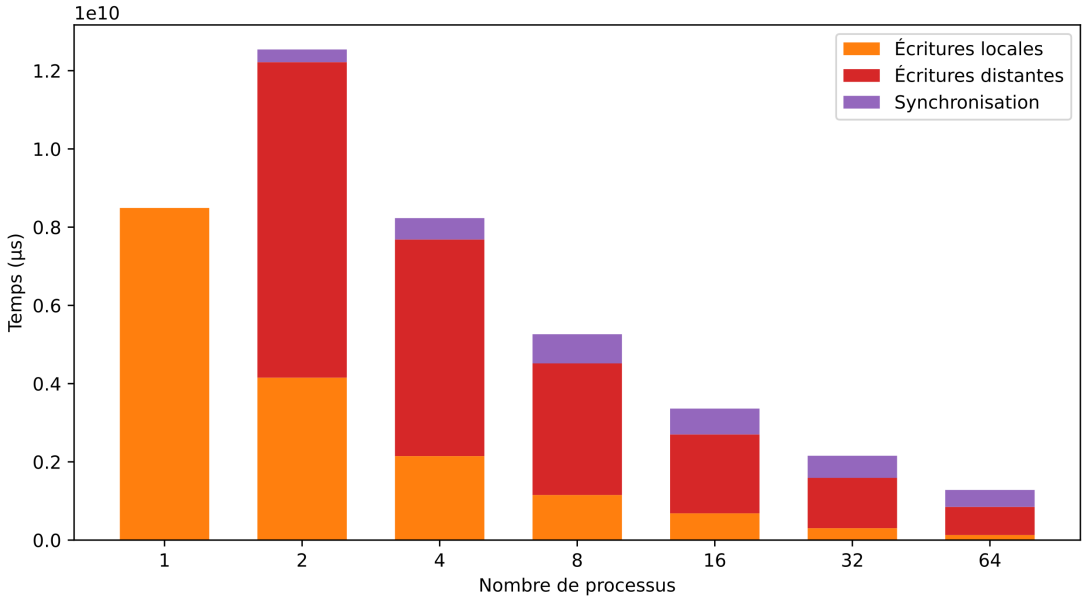
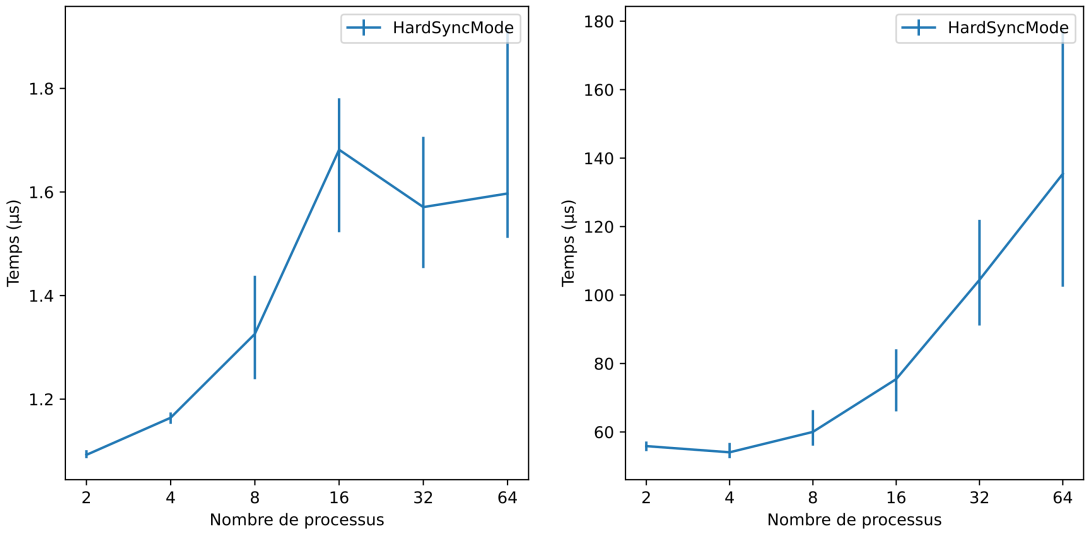


FIGURE 5.11. – Décomposition temporelle des différentes opérations de synchronisation pour un modèle `WRITE_ALL` sur 1000 pas de temps avec le `HardSyncMode`.



(a) Écritures locales.

(b) Écritures distantes.

FIGURE 5.12. – Temps d'écritures individuelles pour un modèle `WRITE_ALL` avec le `HardSyncMode`.

(16 octets). Les lectures et les écritures représentent donc un coût en communication similaire.

Il semble donc que la gestion de la concurrence d'accès plus stricte dans le cas des écritures n'engendre pas une hausse de coût significative par rapport aux lectures. En effet, l'exclusivité des accès s'applique à l'échelle des agents, et non à l'échelle des processus. Par exemple, même si pour 64 processus tous les processus requièrent l'accès exclusif à un agent du processus P, il est peu probable, avec en moyenne 15 625 agents par processus, que plusieurs processus requièrent simultanément l'accès au même agent. Or le processus P peut accorder aux autres processus plusieurs accès exclusifs à des agents différents. La concurrence d'accès en écriture doit donc être assurée dans la théorie, afin d'assurer la cohérence du peu d'écritures réellement concurrentes, mais ne pose pas de fortes contraintes dans la pratique. Cette conclusion n'est pas le fruit d'un biais dû à notre configuration. Le graphe Small-World se veut au contraire représentatif de modèles Multi-Agents réels nécessitant des écritures distantes. En effet, dans le cas d'un modèle réel, et encore davantage si les interactions ont lieu selon des relations de proximité géographique, la probabilité d'écritures réellement concurrentes est faible si le nombre d'agents est largement supérieur au nombre de processus. Nous supposons également une certaine uniformité des interactions : les conclusions seraient autres dans le cas d'un graphe d'interactions avec un haut niveau de centralité.

Nous pouvons augmenter les contraintes de concurrence avec le cas d'un modèle `READ_ALL_WRITE_ONE`, dont les résultats sont présentés sur la figure 5.13. On observe que les performances du `HardSyncMode` restent stables en combinant lectures et écritures. Nous ne présentons pas ici l'évolution des coûts individuels de lectures et d'écritures, car il est difficile dans ce cas de répartir de manière pertinente le temps de synchronisation entre les lectures et les écritures, ce qui rend notamment la formule 5.3 inapplicable en l'état.

Nos expérimentations sur le *Méta-Modèle* permettent d'estimer les performances des algorithmes d'équilibrage de charge et de synchronisation pour des classes de modèles génériques. Nous pouvons dès lors chercher à confirmer les résultats attendus sur un modèle réel. Or des implémentations des comportements d'infection dans un modèle épidémiologique sont assimilables aux comportements `READ_ALL` et `WRITE_ALL`.

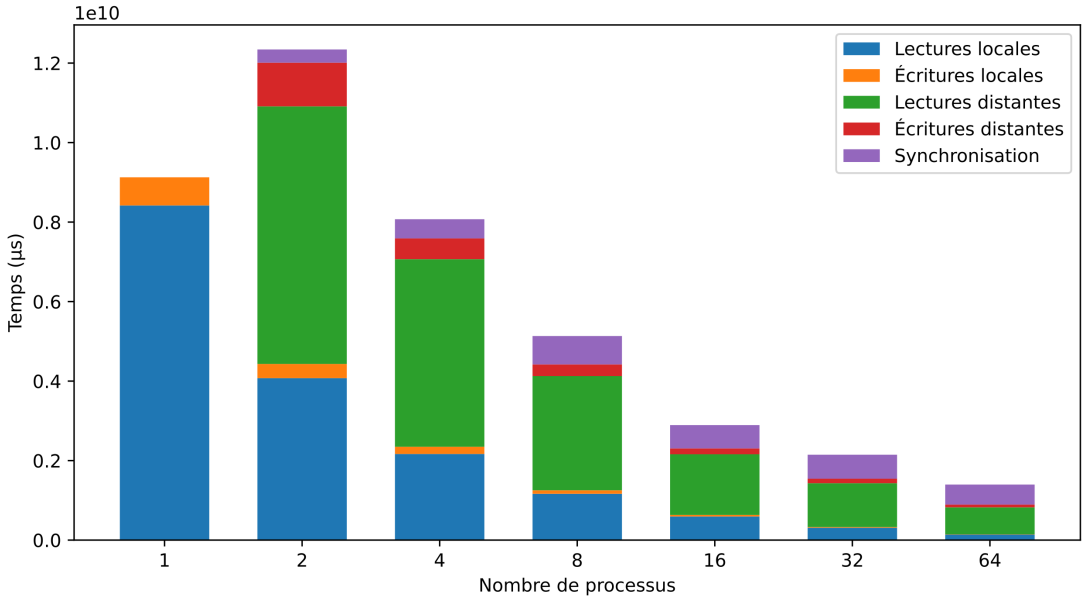


FIGURE 5.13. – Décomposition temporelle des différentes opérations de synchronisation pour un modèle `READ_ALL_WRITE_ONE` sur 1000 pas de temps avec le `HardSyncMode`.

5.5. Impact sur les résultats des modèles

Pour confirmer les performances des modes de synchronisation sur un modèle réel et pour évaluer leur impact sur les résultats des modèles, nous nous affranchissons du *Méta-modèle* au profit de l'analyse d'un modèle épidémiologique. L'implémentation du modèle constitue également une démonstration des capacités de FPMAS et des concepts abstraits introduits jusqu'à présent.

5.5.1. Modèle *Virus*

Nous étudions le cas d'un modèle épidémiologique SIRD (*Susceptible*, *Infecté*, *Rétabli*, *Décédé*, ou *Susceptible*, *Infected*, *Removed*, *Dead* pour la version anglophone), basé sur un environnement à base de grille de taille $X \times Y$ dans lequel N agents, initialisés aléatoirement et uniformément sur l'environnement, se déplacent aléatoirement dans leur voisinage de Moore à chaque pas de temps. À l'initialisation, n_i agents sont sélectionnés aléatoirement et passent à l'état *Infecté*. Le modèle est simulé pendant T pas de temps.

À chaque pas de temps, les agents *Infectés* peuvent guérir et passer à l'état *Rétabli* avec une probabilité γ, mourir et passer à l'état *Décédé* avec une probabilité μ, ou infecter chaque agent voisin *Susceptible* avec une probabilité α.

Telle est la spécification du modèle *Virus*, indépendamment de l'environnement de simulation considéré, séquentiel ou distribué.

Paramètre	Valeur
$X \times Y$	1500×1500
N	1 000 000
n_i	10
T	1000
γ	0,035
μ	0,005
α	0,25

TABLE 5.6. – Paramètres d'expérimentation du modèle *Virus*.

Il est possible d'implémenter ce modèle de manière distribuée dans FPMAS à partir des interfaces de lecture et d'écriture. Le mécanisme d'infection décrit peut en particulier être implémenté de deux manières différentes :

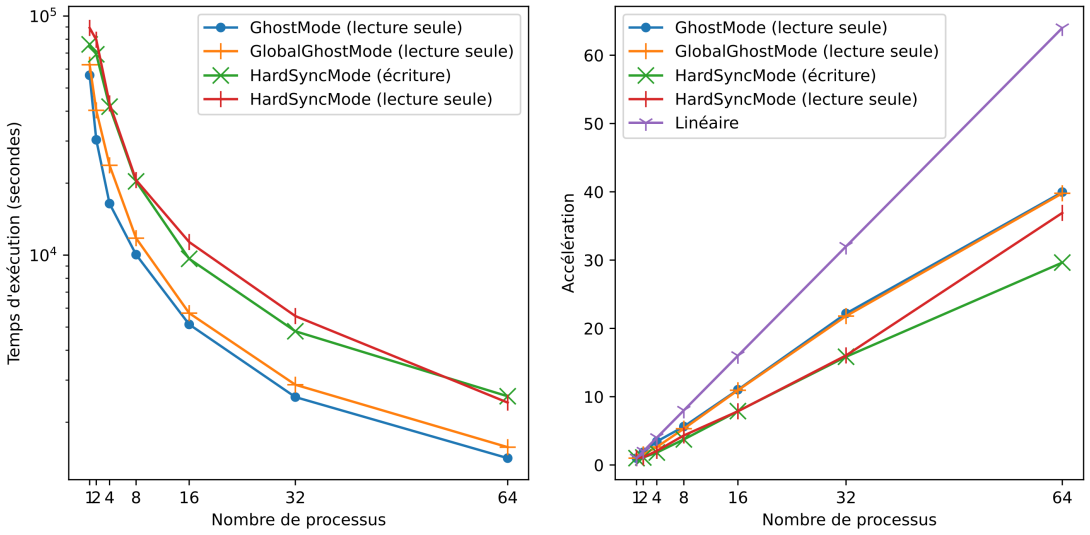
— Version en *lecture seule* : les agents *Susceptibles* réalisent une expérience de Bernoulli de paramètre β pour chaque voisin *Infecté*. Si le résultat d'au moins une est positif, l'agent devient *Infecté*.
— Version avec *écritures* : les agents *Infectés* infectent directement leurs voisins susceptibles en changeant leur état avec une probabilité β.

Conformément aux propriétés des modes de synchronisation implémentés dans FPMAS, la version avec écritures ne peut s'exécuter sans violation des règles du modèle qu'avec le `HardSyncMode`. En revanche, le `GhostMode`, le `GlobalGhostMode` et le `HardSyncMode` peuvent être utilisés pour simuler le modèle en lecture seule. À noter que la version en lecture seule correspond à un comportement de type `READ_ALL`, et que la version avec écritures correspond à un comportement `WRITE_ALL`. Chaque écriture, qui inclut par défaut la lecture des données actuelles, correspond alors à la vérification de l'état susceptible des agents et à leur infection éventuelle. Les résultats obtenus avec le *Méta-Modèle* pour ces types de comportements (figure 5.7 et 5.11) permettent déjà de prédire dans une certaine mesure les performances attendues pour chaque version du modèle épidémiologique.

Les expériences ont été menées avec FPMAS 1.6 [2] et la version 1.1 de l'implémentation du modèle *Virus* [5]. L'intégralité des données brutes sont accessibles librement et de manière pérenne grâce à la plateforme dat@UBFC [28].

5.5.2. Performances

Nous étudions d'abord les performances de FPMAS dans le cadre de la simulation du modèle *Virus*, en faisant varier le nombre de processus. Les paramètres du modèle sont donnés dans le tableau 5.6. Ces paramètres ont été choisis empiriquement pour faire apparaitre la dynamique classique des courbes de résultats des modèles SIRD tout en étalant la courbe d'infection sur les 1000 pas de temps de la simulation, mais aussi pour assurer un nombre d'agents par processus décent même sur 64 processus, tout en gardant une taille de modèle compatible avec une exécution sur un seul processus.



(a) Temps d'exécution

(b) Accélération par rapport à l'utilisation d'un seul processus

FIGURE 5.14. – Performances de la simulation du modèle virus

Les résultats de performances pour les modes de synchronisation considérés sont donnés dans la figure 5.14. La figure 5.14a montre les temps d'exécution pour chaque nombre de processus considéré, avec une échelle logarithmique en ordonnée. Chaque valeur est obtenue par la moyenne de 10 expériences avec des graines aléatoires différentes. La figure 5.14b montre l'*accélération* (ou « speedup ») pour chaque nombre de processus, définit comme $S_p = T_0/T_p$ où T_p désigne le temps d'exécution sur p processus. La courbe linéaire montre l'accélération idéale, pour laquelle l'exécution sur p processus divise le temps d'exécution par p par rapport à une exécution séquentielle.

On observe tout d'abord qu'en termes de temps d'exécution, les résultats obtenus sont globalement cohérents avec les résultats présentés dans la section 5.4.2, en particulier ceux relatifs au modèle `READ_ALL` auquel correspond le modèle *Virus* en lecture seule. Les différences s'expliquent d'une part par l'utilisation d'un modèle spatial, impliquant une structure globale de graphe dynamique, et d'autre part par le comportement réel des agents. On retrouve cependant la tendance générale des performances relatives des différents modes de synchronisation :

— Le mode le plus performant est le `GhostMode`.
— Le `GlobalGhostMode` est légèrement plus lent, en raison de la gestion des copies *ghost*.
— Le `HardSyncMode` est le plus coûteux, en raison de l'utilisation de communications point à point et de l'importation systématique des données des agents.

On pourrait s'étonner de l'équivalence des performances du `HardSyncMode` pour les modèles en lecture seule et avec écritures. Ces résultats confirment cependant les conclusions déjà établies à la section 5.4.3, selon laquelle les coûts des lectures et des

écritures individuelles dans les modèles `READ_ALL` et `WRITE_ALL` sont similaires, en raison de la faible concurrence d'accès réelle à chaque agent.

Pour finir, on observe dans la figure 5.14b que tous les modes font preuve d'une accélération acceptable. On peut noter la présence atypique d'une hausse supra-linéraire de l'accélération avec le `HardSyncMode` sur le modèle en lecture seule de 32 à 64 processus : le nombre de processus double, et pourtant $S_{64}/S_{32} = T_{32}/T_{64} > 2$. Cette hausse peut s'expliquer par le découpage à base de grille et par la distribution des communications elles-mêmes : avec un découpage sur 64 processus, même si le nombre global de communications augmente, le nombre de communications de chaque processus avec ses voisins diminue.

Même si l'analyse sur d'autres modèles seraient pertinente, nous considérons que l'analyse générique de performances permise par le *Méta-Modèle* et sa confirmation par les résultats du modèle *Virus* valident en grande partie les performances des modes de synchronisation implémentés dans FPMAS. Nous poursuivons l'étude expérimentale des modes de synchronisation avec l'influence des modes sur la reproductibilité réelle des simulations.

5.5.3. Reproductibilité

Nous proposons ici une analyse expérimentale de la reproductibilité effective des modes `GhostMode` et `HardSyncMode` dans le cas du modèle *Virus*.

Il a déjà été démontré théoriquement à la section 5.3 que le `GlobalGhostMode` est capable d'atteindre le niveau de reproductibilité le plus haut, qui garantit une reproductibilité stricte indépendamment du partitionnement et donc du nombre de processus. Des résultats expérimentaux ont validé la capacité d'atteindre ce niveau de reproductibilité avec FPMAS.

Le `HardSyncMode` n'offre théoriquement aucune garantie de reproductibilité, et le `GhostMode` se limite à la reproductibilité stricte à partitionnement et ordre d'exécution des agents fixés. Nous quantifions ici la capacité de ces modes à dépasser ces limites dans la pratique.

Nous basons l'étude de la reproductibilité sur le cas de référence de la figure 5.15, obtenu par la simulation d'un modèle *Virus* avec les mêmes paramètres que précédemment (tableau 5.6) en `GlobalGhostMode` avec 10 graines différentes. Les résultats de chaque expérience, représentés superposés sur la figure, sont obtenus avec 64 processus, mais ne dépendent pas du nombre de processus. Ce mode atteignant le niveau de reproductibilité le plus élevé, la variabilité entre les courbes ne dépend que de la graine aléatoire, d'où son utilisation comme référence.

Nous étudions d'abord le cas du `HardSyncMode`. La figure 5.16 montre les résultats de 10 expérimentations avec les mêmes paramètres que pour la figure 5.15, mais avec le `HardSyncMode`, 64 processus, et une graine aléatoire fixée. On observe donc ici la variabilité propre au `HardSyncMode` entre plusieurs exécutions à partitionnement et ordre d'exécution fixés sur 64 processus. Or cette variabilité est clairement négligeable par rapport à celle liée à la graine aléatoire (figure 5.15), dans la version en lecture seule (figure 5.16a) comme dans la version avec écritures (figure 5.16b).

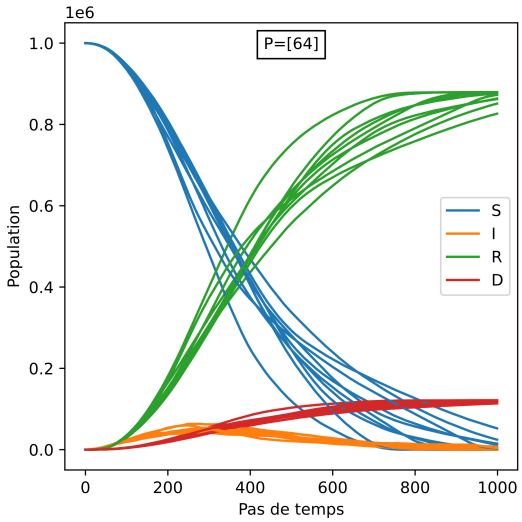


FIGURE 5.15. – Analyse de la reproductibilité du `GlobalGhostMode` avec 10 graines différentes (lecture seule), utilisée comme cas de référence.

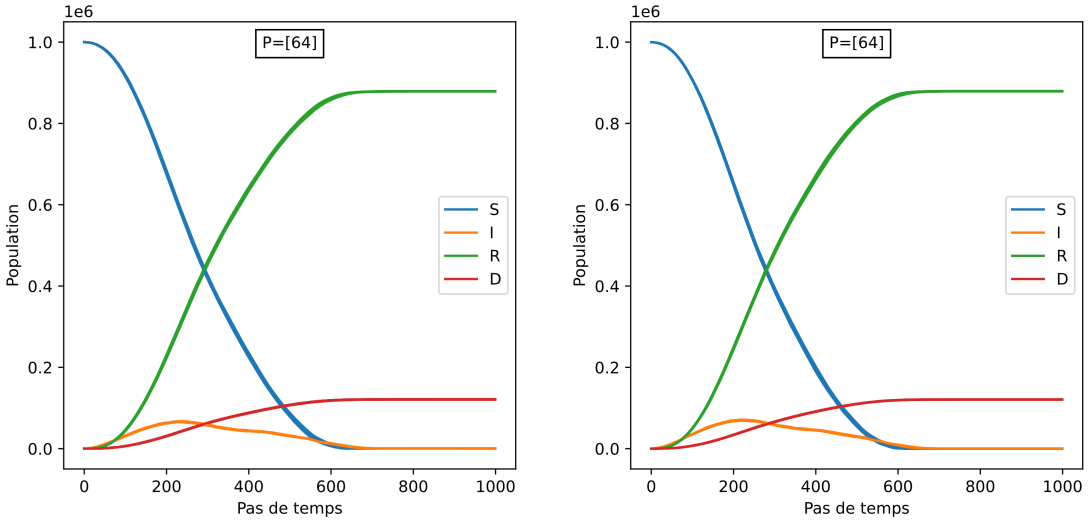
On en déduit que même si le `HardSyncMode` n'offre aucune garantie de reproductibilité stricte à partitionnement et ordre d'exécution fixe, celui-ci atteint dans la pratique une forme de reproductibilité statistique dans les mêmes conditions, au moins dans le cas du modèle *Virus*.

Nous étudions ensuite la variabilité liée à l'utilisation du `GhostMode` en fonction du nombre de processus. La superposition des courbes de la figure 5.17 est cette fois obtenue avec les résultats d'expérience sur différents nombres de processus, donné par l'ensemble P. Chaque courbe est obtenue par la moyenne de 10 expériences avec des graines différentes pour chaque nombre de processus [2]. En effet, utiliser la même graine avec des nombres de processus différents n'a que très peu de sens, car les séquences de nombre générés sur chaque processus par la méthode présentée à la section 3.5.2 seront fondamentalement différentes. Effectuer la moyenne sur 10 expériences permet de limiter la variabilité liée à la génération de nombres aléatoires, pour se concentrer sur celle liée au nombre de processus.

Là encore, on observe que la variabilité de la figure 5.17 est négligeable par rapport au cas de référence. Le `GhostMode` permet donc d'atteindre une forme de reproductibilité statistique indépendante du nombre de processus, et donc potentiellement indépendante du partitionnement, au moins dans le cas du modèle *Virus*.

Même si une étude plus fine et extensive de ces phénomènes reste nécessaire, on constate que le `GhostMode` et le `HardSyncMode` peuvent atteindre dans la pratique des niveaux de reproductibilité statistique qui dépassent le niveau de reproductibilité stricte équivalent qui leur a été assigné à la section 5.3. De tels niveaux de reproductibilité statistique peuvent s'avérer largement suffisants pour l'utilisateur final, même si une reproductibilité

2. Les 10 mêmes graines que pour le cas de référence sont systématiquement utilisées.



(a) `HardSyncMode` avec la même graine sur 10 itérations (lecture seule).

(b) `HardSyncMode` avec la même graine sur 10 itérations (écritures).

FIGURE 5.16. – Analyse de la reproductibilité du `HardSyncMode` entre des itérations avec la même configuration.

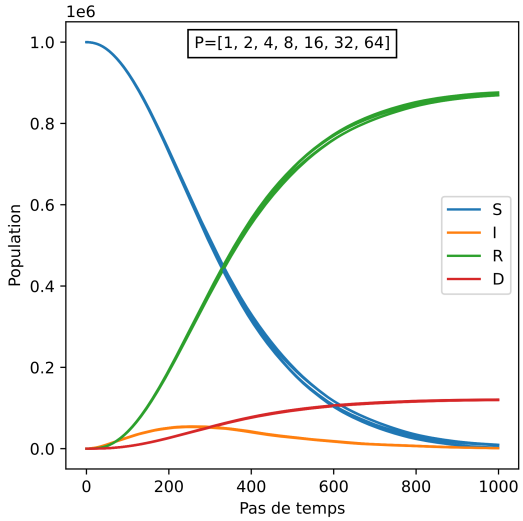


FIGURE 5.17. – Analyse de la reproductibilité du `GhostMode` pour différents nombres de processus.

stricte n'est pas garantie.

5.5.4. Influence de la gestion des lectures et écritures

Nous terminons cette série d'expérimentations par une étude de l'impact des règles de gestion des lectures et écritures définies par chaque mode de synchronisation sur les résultats des modèles. En effet, les modes `GhostMode`, `GlobalGhostMode` et `HardSyncMode` sont utilisables pour simuler le modèle *Virus* en lecture seule ou avec écritures, sans violer les règles du modèle. L'utilisation de chaque mode donne cependant lieu à différentes gestions des lectures et écritures, conformément à ce qui a été présenté dans le tableau 5.1.

La figure 5.18 présente des résultats obtenus avec chaque mode sur 64 processus. Comme précédemment, chaque courbe correspond à la moyenne de 10 itérations avec des graines différentes, afin de limiter les causes de variabilité autres que celles dues aux différentes gestions des lectures et écritures par les modes de synchronisation. On constate cette fois une variabilité non négligeable entre certains modes par rapport à la variabilité de référence de la figure 5.15, notamment au niveau des courbes *Removed* et *Infected*.

L'avance des courbes en `HardSyncMode` pour le modèle avec écritures est synonyme d'une propagation plus rapide de l'infection. En effet, dans cette configuration, l'exécution du comportement d'un agent peut donner lieu à l'infection de plusieurs autres agents au temps T, y compris s'ils sont localisés sur d'autres processus. Si les agents infectés au temps T n'ont pas encore été exécutés, ils pourront à leur tour infecter d'autres agents au temps T.

En comparaison, la courbe avec le `HardSyncMode` en lecture seule est plus lente, car même si la lecture de l'état infecté des voisins a lieu au temps T, au plus un agent (l'agent lui-même) peut être infecté lors de l'exécution du comportement d'un agent. Les résultats ainsi obtenus sont similaires à ceux du `GhostMode`, avec une très légère avance pour le `HardSyncMode`. Le fait de lire systématiquement l'état des agents *distants* au temps T depuis les processus distants n'a donc pas un impact significatif sur les résultats du modèle, et peut représenter un coût inutile dans ce cas. Ce constat peut s'expliquer par la faible proportion d'infections ayant lieu entre des agents exécutés sur des processus différents.

Pour finir, la courbe d'infection avec le `GlobalGhostMode` est retardée par rapport aux autres modes. En effet, la totalité des lectures s'effectuant à $T - 1$, y compris sur les agents *locaux*, les agents infectés au temps T ne peuvent transmettre le virus qu'à partir du pas de temps suivant.

Or aucune de ces expérimentations ne viole les spécifications du modèle d'origine, données à la section 5.5.1. Il est donc clair que les modalités d'interactions imposées par les modes de synchronisation, ou plus particulièrement les adaptations du modèle d'origine à ces modalités, peuvent avoir un impact sur les résultats des modèles. Selon les besoins de l'utilisateur, cet impact peut être significatif ou non. Par exemple, dans le cas du modèle *Virus*, la variation de la hauteur du pic d'infection observée avec les différents modes peut être considérée comme une variable critique.

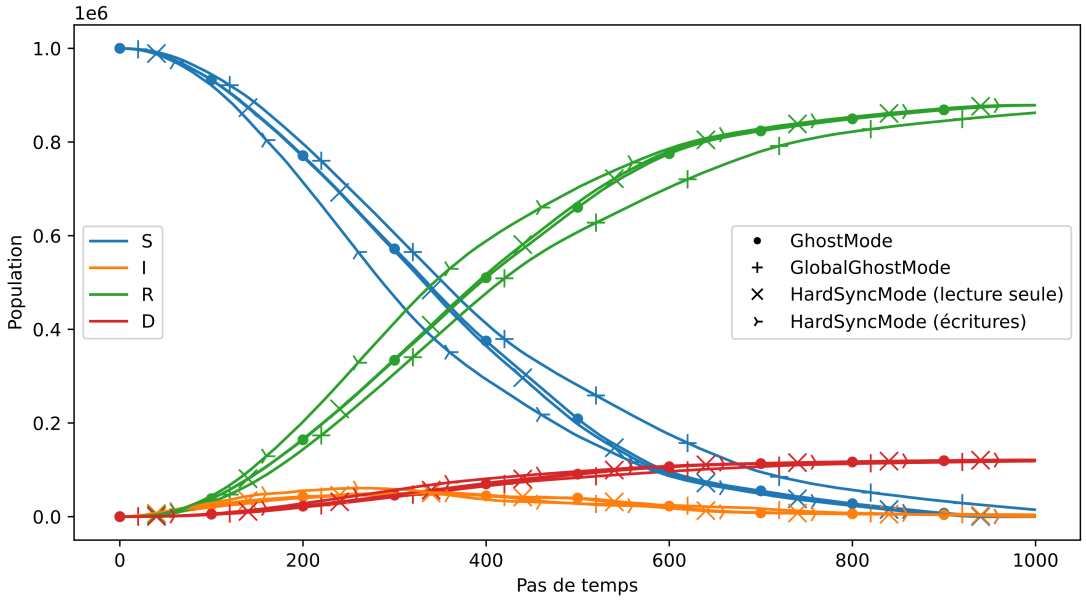


FIGURE 5.18. – Comparaison des résultats obtenus avec différents modes de synchronisation

5.6. Synthèse

Dans ce chapitre, nous avons d'abord présenté les enjeux liés à la gestion des lectures et des écritures distantes dans la simulation distribuée de SMA, notamment en termes de temporalité des interactions. D'où la définition d'une interface générique permettant d'implémenter divers modes de synchronisation des données, ainsi que la spécification de plusieurs modes, dont trois ont été implémentés dans FPMAS. Nous avons ensuite mené une étude extensive de ces modes dans la suite du chapitre. Une étude théorique a d'abord été réalisée, avec une comparaison des modes selon les possibilités d'interactions entre agents distants, et selon les garanties de reproductibilité. Une première étude expérimentale basée sur le *Méta-Modèle* a ensuite permis de comparer de manière générique les modes de synchronisation implémentés dans FPMAS. Grâce à une seconde étude expérimentale basée sur le modèle *Virus*, nous avons pu confirmer les performances des modes de synchronisation sur un modèle réel, et obtenir des résultats préliminaires sur les niveaux de reproductibilité effectifs de ces modes et sur leur impact sur les résultats des modèles.

Il apparait alors que chaque mode possède des avantages spécifiques par rapport aux autres, résumés dans le tableau 5.7 pour les modes implémentés. Le `HardSyncMode` est le plus efficace en termes d'interactions, car il est le seul à permettre les écritures cohérentes au temps T entre les processus. Ce mode nécessite donc peu d'adaptation des modèles à l'environnement distribué, en comparaison des autres modes qui permettent seulement les lectures. Le `GhostMode` est le plus performant, car il nécessite peu de communications

Mode	Interactions	Performances	Reproductibilité
`GhostMode`	+	+++	++
`GlobalGhostMode`	+	++	+++
`HardSyncMode`	+++	+	+

TABLE 5.7. – Comparaison qualitative des modes de synchronisation implémentés dans FPMAS selon différents critères.

collectives et doit gérer un minimum de copies *ghost*. Le `GlobalGhostMode` est enfin le seul mode permettant d'atteindre la reproductibilité stricte indépendamment du partitionnement et du nombre de processus.

Le choix du mode de synchronisation est donc nécessairement le fruit d'un compromis issu des besoins de l'utilisateur final. D'où l'intérêt de proposer des plateformes de simulation distribuée de SMA permettant la mise en place de différents modes de synchronisation.

Chapitre 6.

Conclusion

Ce dernier chapitre est l'occasion de dresser un bilan de nos contributions au domaine de la simulation distribuée de SMA, ainsi que leurs perspectives.

Bilan

Le chapitre 1 a été l'occasion de définir les problématiques relatives au contexte de notre étude. Nous pouvons alors dresser le bilan de nos travaux au regard de nos questions de recherches.

Quelles sont les difficultés à surmonter pour permettre la simulation distribuée de SMA ?

Notre propre expérience de développement de la plateforme FPMAS ainsi que l'étude de l'état de l'art nous ont permis d'isoler des problèmes communs à toute distribution de simulation de SMA, notamment la *distribution des SMA*, l'*équilibrage de charge* et la *synchronisation des données*. La synthèse des plateformes RepastHPC, D-MASON, Pandora et FLAME présentée au chapitre 2 selon ces axes de réflexion, même s'ils ne sont souvent abordés que de manière implicite par leurs concepteurs, tend à justifier la généricité de notre approche. De plus, nous avons pu fournir dans les chapitres 3, 4 et 5 une analyse détaillée de ces problèmes sous forme de composants génériques à implémenter, en montrant comment les solutions mises en place par les plateformes existantes pouvaient s'inclure dans l'architecture logicielle générique proposée. L'aboutissement de ces travaux sur une plateforme C++ de simulation distribuée de SMA fonctionnelle, robuste et efficace montre que la résolution des problèmes présentés n'est pas seulement nécessaire mais également suffisante pour mettre en place la simulation distribuée de tout SMA générique.

Quel est le niveau de satisfaction des objectifs d'ergonomie, d'efficacité et de validité des résultats atteint par les solutions proposées dans les plateformes de simulation distribuées existantes ? Quelles sont les limitations selon les types de modèles ?

L'efficacité des plateformes est généralement garantie par leur concepteur, sans quoi leur existence n'aurait que peu d'utilité dans le contexte de la simulation distribuée.

Nous avons cependant pu constater de grandes disparités en termes d'ergonomie et de validité des résultats. En effet, une des limitations des plateformes existantes réside dans la nécessité de compétences en parallélisme pour les utiliser. Nous avons non seulement fait le choix d'abstraire au maximum les problématiques liées à la distribution, mais aussi montré qu'il est possible de concevoir une plateforme de simulation distribuée respectant ce principe grâce aux interfaces introduites dans la partie II et à l'exemple d'implémentation de FPMAS.

La quantité de travail nécessaire pour adapter un modèle existant à l'exécution distribuée représente une autre limitation en termes d'ergonomie, ce qui nous mène au problème de validité des résultats. En effet, si l'implémentation distribuée de certains modèles est difficile, elle est parfois impossible sans changer leurs règles ou sans tolérer leur violation, par exemple par la gestion incohérente des interactions entre agents *distants*. Nous avons pu voir au chapitre 2 que les modèles Multi-Agents peuvent exposer différents besoins en termes de *lectures* et d'*écritures*. Or l'analyse de l'état de l'art montre que les solutions existantes imposent aux modèles la lecture seule en mode *ghost* sur tous les agents (D-MASON), seulement sur les agents *distants* (RepastHPC), l'envoi de messages explicites en fin de pas de temps pour les interactions entre agents (FLAME), ou des solutions avec un impact critique sur la distribution et le schéma d'exécution des agents (Pandora). Aucune des solutions étudiées ne permet la réalisation de lectures ou d'écritures concurrentes entre les processus au cours d'un pas de temps, une fonctionnalité pourtant nécessaire à la simulation de nombreux modèles Multi-Agents. Nous avons par ailleurs montré au chapitre 5 l'impact que peut avoir la distribution, et particulièrement la synchronisation des données, sur la reproductibilité de la simulation, qui peut représenter un critère de validité important pour certains utilisateurs. D'où la nécessité de mettre en place des solutions flexibles pouvant adapter l'exécution distribuée à la diversité des modèles Multi-Agents.

Comment mettre en place des solutions adaptables à tout type de modèle pour atteindre ces objectifs ?

Les expérimentations menées aux chapitres 3 et 5 montrent la nécessité d'adapter les techniques d'équilibrage de charge et de synchronisation des données à la diversité des modèles Multi-Agents et des besoins de l'utilisateur par exemple en termes de performance et de reproductibilité. C'est pourquoi nous défendons la conception par interface des plateformes de simulation distribuées de SMA génériques. Cette conception permet d'apporter simplement des solutions indépendantes aux problèmes précédemment évoqués, comme nous l'avons fait par les algorithmes présentés dans la partie II. Le schéma de distribution proposé au chapitre 3 permet notamment de préserver la structure des modèles indépendamment de leur partitionnement, ce qui permet aux algorithmes d'équilibrage inclus à la plateforme de produire des partitions sans aucune contrainte relative à la distribution du modèle. Les exemples des chapitres 4 et 5 montrent par ailleurs la possibilité d'implémenter des solutions diverses à chaque problème à partir d'une interface commune, sans altérer le reste du système. Il est alors possible de trivialement passer d'un algorithme à l'autre sans altérer l'implémentation des modèles,

ce qui permet à l'utilisateur de facilement choisir les solutions correspondant le mieux à ses besoins, sans pour autant se soucier des détails d'implémentation des composants ou des problèmes liés à la distribution. Les nombreux résultats présentés dans ce document pour le *Méta-Modèle* ou le modèle *Virus* avec divers algorithmes d'équilibrage de charge ou de synchronisation des données ne sont qu'un aperçu de la flexibilité permise par l'architecture logicielle générique utilisée par FPMAS.

Perspectives

Notre étude se focalise avant tout sur la structure de l'architecture logicielle proposée et sur la définition des problèmes à résoudre. Ainsi l'exploration des solutions possibles aux problèmes de la *distribution des SMA*, de l'*équilibrage de charge* et de la *synchronisation* constituent en eux-mêmes des sujets de recherche. L'état de l'art et nos propres exemples d'algorithmes d'équilibrage de charge et de synchronisation des données montrent à la fois la possibilité et la nécessité d'implémenter et d'analyser d'autres solutions.

Même si nous avons étudié les performances des algorithmes de partitionnement de graphe dans le contexte de la simulation distribuée de SMA, il est possible d'intégrer à FPMAS de nombreuses implémentations d'algorithmes d'équilibrage basés sur d'autres concepts ou conçus et analysés spécifiquement pour la simulation distribuée de SMA.

La synchronisation des données offre également de bonnes perspectives, comme les modes définis dans ce document mais n'ayant pas encore été implémentés, ou la conception d'autres modes permettant de satisfaire de nouveaux besoins. Notre étude partielle de la reproductibilité effective dans le cas du modèle *Virus* semble justifier la nécessité d'une étude plus détaillée de l'impact de la synchronisation des données sur la reproductibilité des simulations.

L'applicabilité de la plateforme logicielle proposée à la distribution d'autres modèles ou plateformes de simulation, comme envisagé avec la plateforme GAMA, constitue des opportunités en termes d'exécution large échelle pour la communauté Multi-Agents.

Enfin, de nombreuses optimisations de la plateforme FPMAS sont souhaitables. Si elle permet de résoudre facilement les problèmes de continuité et de synchronisation des données pour tout SMA générique, la structure interne à base de graphe est cependant très limitante en termes de performances, notamment dans le cas des simulations spatiales continues ou à base de grille. Il est donc pertinent d'envisager l'utilisation de structures de données internes plus adaptées au modèle simulé, tout en préservant l'abstraction et la flexibilité de la distribution qui constituent les atouts de FPMAS.

Liste des algorithmes

Liste des logiciels

Les logiciels listés ici sont archivés par Software Heritage [1] et identifiés grâce à un identifiant persistent, le SWHID. Cet identifiant constitue une référence permanente vers les sources des logiciels cités ou utilisés dans ces travaux, indépendamment de l'évolution de leurs dépôts sources (par exemple hébergés sur GitHub ou sur une instance GitLab), qui eux ne fournissent aucune garantie de pérénité. Conformément aux recommendations de Software Heritage:

— le SWHID contient autant d'informations contextuelles que possible.
— seul le SWHID est inclus dans la version imprimable, et pas les URL complets vers Software Heritage, car il n'y a pas non plus de garantie de stabilité du schéma URI du site à long terme.

[1] Paul Breugnot, *FPMAS* 2019. FEMTO-ST. SWHID : ⟨swh:1:dir:c1192b4e6fd5a094ad086bdb9b554bcad4a0906e;origin=https://github.com/FPMAS/FPMAS;visit=swh:1:snp:be7ac44b4d620fe1b5324a23d647c94eabb82b20;anchor=swh:1:rev:044e84a675e6f80d4a21c14769e57e5b6b8fc641⟩.

[2] Paul Breugnot, *FPMAS* version 1.6, 2019. FEMTO-ST. SWHID : ⟨swh:1:rel:782cb0732ac4c66e0db27f3fc9253da3d470b4cb;origin=https://github.com/FPMAS/FPMAS;visit=swh:1:snp:be7ac44b4d620fe1b5324a23d647c94eabb82b20⟩.

[3] Paul Breugnot, *FPMAS MetaModel* version 1.1, 2022. FEMTO-ST. SWHID : ⟨swh:1:rel:403dd6c0fc3ad6266a20347c188ea3665778ea81;origin=https://github.com/FPMAS/fpmas-metamodel;visit=swh:1:snp:2ce461b4cd9c927b9da0eb4946baa6e617e3630e⟩.

[4] Paul Breugnot, *FPMAS ObjectPack* 2019. FEMTO-ST. SWHID : ⟨swh:1:cnt:0bc16fa2d4ce5b93384702c6bc20e5a9c8933e2c;origin=https://github.com/FPMAS/FPMAS;visit=swh:1:snp:be7ac44b4d620fe1b5324a23d647c94eabb82b20;anchor=swh:1:rev:044e84a675e6f80d4a21c14769e57e5b6b8fc641;path=/src/fpmas/io/datapack.h⟩.

[5] Paul Breugnot, *FPMAS Virus Model* version 1.1, 2022. FEMTO-ST. SWHID : ⟨swh:1:rel:564e3f717b327686f3dc213a6b7ac850a4f8ce49;origin=https://github.com/FPMAS/fpmas-virus;visit=swh:1:snp:8285b6abcb436ffc65d74f57f514c012534f0e3b⟩.

1. https://www.softwareheritage.org/

[6] Karen Devin et al., *Zoltan* 2022. Sandia National Laboratories.
URL : https://sandialabs.github.io/Zoltan/(visité le 12/10/2022), SWHID : ⟨swh:1:dir:2cf75d4a15c616741c455d00436e9c4c0f1463f0;origin=https://github.com/sandialabs/Zoltan;visit=swh:1:snp:c1f903fb0092cd0e7847932284042f5f982efce2;anchor=swh:1:rev:f6361719dd66cac62db8dbed120704e436a5ee81⟩.

[7] Niels Lohmann, *Json for Modern C++* 2022. SWHID : ⟨swh:1:dir:4c18fc5d124bd70a990f09524cae031967ef01ce;origin=https://github.com/nlohmann/json;visit=swh:1:snp:19a433a9ec78180ed98b31e8dae5216c5db5bfd9;anchor=swh:1:rev:a3e6e26dc83a726b292f5be0492fcc408663ce55⟩.

[8] Tiago P. Peixoto, *Python Graph-Tool* version 2.45, mai 2022.
URL : https://graph-tool.skewed.de/(visité le 27/10/2022), SWHID : ⟨swh:1:rel:314dff4d8c8f7711eb6975468c6a3ecbdc76e1d4;origin=https://git.skewed.de/count0/graph-tool;visit=swh:1:snp:27454cfe8416332b1d1fce9b491a61f25d6a71a5⟩.

[9] Robert Ramey, *Boost Serialization Library* 2004. Boost.
URL : https://www.boost.org/doc/libs/1_80_0/libs/serialization/doc/index.html(visité le 25/10/2022), SWHID : ⟨swh:1:dir:0421ff4708cc7c1766ccc9c2bc095ec6298e324e;origin=https://github.com/boostorg/serialization;visit=swh:1:snp:d73d1fcca6866709e759d8532a04e82fcac0a986;anchor=swh:1:rev:3f322d4adc3c88a667751ad66ce19217a3bba1f9⟩.

[10] Uri Wilensky, *NetLogo* 1999.
Center for Connected Learning and Computer-Based Modeling.
URL : http://ccl.northwestern.edu/netlogo/(visité le 18/10/2022), SWHID : ⟨swh:1:dir:1bf75ffd6af135d31d28fa66cf66390d26b4e2dd;origin=https://github.com/NetLogo/NetLogo;visit=swh:1:snp:67e0601fe0c4e88d119941c41f407ad50c9410cf;anchor=swh:1:rev:ce236224a4bf88f2b08c17a92fe3a8341de6a720⟩.

[11] Uri Wilensky, *NetLogo Flocking Model* 1998.
Center for Connected Learning and Computer-Based Modeling.
URL : http://ccl.northwestern.edu/netlogo/models/Flocking(visité le 18/10/2022), SWHID : ⟨swh:1:cnt:3f43029c8fed33edd254cfa53f801b2186dceb9e;origin=https://github.com/NetLogo/models;visit=swh:1:snp:143b8edfef5d7c73bc29fb7ff7677b58a1688e09;anchor=swh:1:rev:f5d3d913e75cf7c1128f24978893734bea63a3a7;path=/Sample%20Models/Biology/Flocking.nlogo⟩.

[12] Uri Wilensky, *NetLogo Models Library* 1999.
Center for Connected Learning and Computer-Based Modeling.
URL : http://ccl.northwestern.edu/netlogo/models/(visité le 18/10/2022), SWHID : ⟨swh:1:dir:60ec979000826ef87ee8fc836695a81be81304c4;origin=https://github.com/NetLogo/models;visit=swh:1:snp:143b8edfef5d7c73bc2

9fb7ff7677b58a1688e09;anchor=swh:1:rev:f5d3d913e75cf7c1128f24978893734bea63a3a7⟩.

[13] Uri Wilensky, *NetLogo Virus Model* 1998.
Center for Connected Learning and Computer-Based Modeling.
URL : http://ccl.northwestern.edu/netlogo/models/Virus(visité le 18/10/2022), SWHID : ⟨swh:1:cnt:36885749060bdb7e52b9f7c861268419486995 75;origin=https://github.com/NetLogo/models;visit=swh:1:snp:143b8edf ef5d7c73bc29fb7ff7677b58a1688e09;anchor=swh:1:rev:f5d3d913e75cf7c112 8f24978893734bea63a3a7;path=/Sample%20Models/Biology/Virus.nlogo⟩.

[14] Uri Wilensky, *NetLogo Wolf Sheep Predation Model* 1997.
Center for Connected Learning and Computer-Based Modeling. URL :
http://ccl.northwestern.edu/netlogo/models/WolfSheepPredation(visité le 18/10/2022), SWHID :
⟨swh:1:cnt:de69ddb21d1388618630e7ddc16899fd342851fe;origin=https://g ithub.com/NetLogo/models;visit=swh:1:snp:143b8edfef5d7c73bc29fb7ff76 77b58a1688e09;anchor=swh:1:rev:f5d3d913e75cf7c1128f24978893734bea63a 3a7;path=/Sample%20Models/Biology/Wolf%20Sheep%20Predation.nlogo⟩.

Bibliographie

[15] Hafiz Usman Ahmed, Ying Huang et Pan Lu.
« A Review of Car-Following Models and Modeling Tools for Human and Autonomous-Ready Driving Behaviors in Micro-Simulation ».
In : *Smart Cities* 4.1 (1 mars 2021), p. 314-335. ISSN : 2624-6511.
DOI : 10.3390/smartcities4010019.

[16] Jérémy Albouys et al.
« SMACH : Multi-agent Simulation of Human Activity in the Household ».
In : *Advances in Practical Applications of Survivable Agents and Multi-Agent Systems : The PAAMS Collection*. Sous la dir. d'Yves Demazeau et al.
Lecture Notes in Computer Science.
Cham : Springer International Publishing, 2019, p. 227-231.
ISBN : 978-3-030-24209-1. DOI : 10.1007/978-3-030-24209-1_19.

[17] Jérémy Albouys Perrois et al.
« Étude de différentes configurations d'autoconsommation collective de l'énergie à l'échelle du quartier à l'aide de la simulation multi-agent ».
In : *28th Journées Francophones sur les Systèmes Multi-Agents (JFSMA)*.
Angers, France, 2020. URL :
https://hal.archives-ouvertes.fr/hal-03195521 (visité le 15/07/2021).

[18] George Almasi. « PGAS (Partitioned Global Address Space) Languages ».
In : *Encyclopedia of Parallel Computing*. Sous la dir. de David Padua.
Boston, MA : Springer US, 2011, p. 1539-1545. ISBN : 978-0-387-09766-4.
DOI : 10.1007/978-0-387-09766-4_210.

[19] Robert Axelrod et William D. Hamilton. « The Evolution of Cooperation ».
In : *Science* (27 mars 1981). DOI : 10.1126/science.7466396.

[20] Mohamed Azeroual et al. « Simulation Tools for a Smart Grid and Energy Management for Microgrid with Wind Power Using Multi-Agent System ».
In : *Wind Engineering* 44.6 (1er déc. 2020), p. 661-672. ISSN : 0309-524X.
DOI : 10.1177/0309524X19862755.

[21] Arnaud Banos et al. « Coupling Micro and Macro Dynamics Models on Networks : Application to Disease Spread ».
In : *Multi-Agent Based Simulation XVI*.
Sous la dir. de Benoit Gaudou et Jaime Simão Sichman.
Lecture Notes in Computer Science.
Cham : Springer International Publishing, 2016, p. 19-33.
ISBN : 978-3-319-31447-1. DOI : 10.1007/978-3-319-31447-1_2.

[22] Eric Blanchart et al. « SWORM : An Agent-Based Model to Simulate the Effect of Earthworms on Soil Structure ».
In : *European Journal of Soil Science* 60.1 (2009), p. 13-21. ISSN : 1365-2389. DOI : 10.1111/j.1365-2389.2008.01091.x.

[23] Jim Blythe et Alexey Tregubov.
« FARM : Architecture for Distributed Agent-Based Social Simulations ».
In : *Massively Multi-Agent Systems II*. Sous la dir. de Donghui Lin et al. Lecture Notes in Computer Science.
Cham : Springer International Publishing, 2019, p. 96-107.
ISBN : 978-3-030-20937-7. DOI : 10.1007/978-3-030-20937-7_7.

[24] Luciano Bononi et al. « A New Adaptive Middleware for Parallel and Distributed Simulation of Dynamically Interacting Systems ». In : *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications.* Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications. Oct. 2004, p. 178-187. DOI : 10.1109/DS-RT.2004.3.

[25] Francisco Borges et al. « Care HPS : A High Performance Simulation Tool for Parallel and Distributed Agent-Based Modeling ».
In : *Future Generation Computer Systems* 68 (1er mars 2017), p. 59-73.
ISSN : 0167-739X. DOI : 10.1016/j.future.2016.08.015.

[26] Francisco Borges et al. « Strip Partitioning for Ant Colony Parallel and Distributed Discrete-event Simulation ».
In : *Procedia Computer Science*. International Conference On Computational Science, ICCS 2015 51 (1er jan. 2015), p. 483-492. ISSN : 1877-0509.
DOI : 10.1016/j.procs.2015.05.272.

[27] Paul Breugnot.
Detailed Execution Times of Several FPMAS Meta-model Instances.
29 mars 2023. DOI : 10.25666/DATAUBFC-2023-04-11-03.

[28] Paul Breugnot.
Execution Times and Results of Instances of an FPMAS Epidemiological Model.
29 mars 2023. DOI : 10.25666/DATAUBFC-2023-04-11-02.

[29] Paul Breugnot et al.
« A Synchronized and Dynamic Distributed Graph Structure to Allow the Native Distribution of Multi-Agent System Simulations ».
In : *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. PDP 2021. Valladolid, Spain, mars 2021, p. 54-61. DOI : 10.1109/PDP52278.2021.00017.

[30] Paul Breugnot et al.
« Data Synchronization in Distributed Simulation of Multi-Agent Systems ».
In : *Advances in Practical Applications of Agents, Multi-Agent Systems, and Complex Systems Simulation. The PAAMS Collection*. PAAMS 2022.
Sous la dir. de Frank Dignum et al. Lecture Notes in Computer Science.

Cham : Springer International Publishing, 2022, p. 50-62.
ISBN : 978-3-031-18192-4. DOI : 10.1007/978-3-031-18192-4_5.

[31] Benjamin BROCK, Aydın BULUÇ et Katherine YELICK.
« BCL : A Cross-Platform Distributed Data Structures Library ».
In : *Proceedings of the 48th International Conference on Parallel Processing.*
ICPP 2019.
New York, NY, USA : Association for Computing Machinery, 5 août 2019,
p. 1-10. ISBN : 978-1-4503-6295-5. DOI : 10.1145/3337821.3337912.

[32] Aydın BULUÇ et al. « Recent Advances in Graph Partitioning ».
In : *Algorithm Engineering : Selected Results and Surveys.*
Sous la dir. de Lasse KLIEMANN et Peter SANDERS.
Lecture Notes in Computer Science.
Cham : Springer International Publishing, 2016, p. 117-158.
ISBN : 978-3-319-49487-6. DOI : 10.1007/978-3-319-49487-6_4.

[33] Umit V. CATALYUREK et Cevdet AYKANAT. « Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication ». In : *IEEE Transactions on Parallel and Distributed Systems* 10.7 (juill. 1999), p. 673-693.
ISSN : 1558-2183. DOI : 10.1109/71.780863.

[34] Umit V. CATALYUREK et al.
« A Repartitioning Hypergraph Model for Dynamic Load Balancing ».
In : *Journal of Parallel and Distributed Computing* 69.8 (août 2009), p. 711-724.
ISSN : 07437315. DOI : 10.1016/j.jpdc.2009.04.011.

[35] Umit V. CATALYUREK et al. « Hypergraph-Based Dynamic Load Balancing for Adaptive Scientific Computations ».
In : *2007 IEEE International Parallel and Distributed Processing Symposium.*
2007 IEEE International Parallel and Distributed Processing Symposium.
Mars 2007, p. 1-11. DOI : 10.1109/IPDPS.2007.370258.

[36] Sheryl L. CHANG et al. « Modelling Transmission and Control of the COVID-19 Pandemic in Australia ».
In : *Nature Communications* 11.1 (1 11 nov. 2020), p. 5710. ISSN : 2041-1723.
DOI : 10.1038/s41467-020-19393-6.

[37] Philippe CHARLES et al.
« X10 : An Object-Oriented Approach to Non-Uniform Cluster Computing ».
In : *ACM SIGPLAN Notices* 40.10 (12 oct. 2005), p. 519-538. ISSN : 0362-1340.
DOI : 10.1145/1103845.1094852.

[38] Cédric CHEVALIER et François PELLEGRINI.
« PT-Scotch : A Tool for Efficient Parallel Graph Ordering ».
In : *Parallel Computing.* Parallel Matrix Algorithms and Applications 34.6
(1^{er} juill. 2008), p. 318-331. ISSN : 0167-8191.
DOI : 10.1016/j.parco.2007.12.001.

[39] Silvano Cincotti, Marco Raberto et Andrea Teglio. « Credit Money and Macroeconomic Instability in the Agent-based Model and Simulator Eurace ». In : *Economics* 4.1 (1er déc. 2010). issn : 1864-6042. doi : 10.5018/economics-ejournal.ja.2010-26.

[40] Claudio Cioffi-Revilla. *A Methodology for Complex Social Simulations.* SSRN Scholarly Paper ID 2291156. Rochester, NY : Social Science Research Network, 1er jan. 2010. doi : 10.2139/ssrn.2291156.

[41] Simon Coakley et al. « Exploitation of High Performance Computing in the FLAME Agent-Based Simulation Framework ». In : *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems.* 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems. Juin 2012, p. 538-545. doi : 10.1109/HPCC.2012.79.

[42] Nicholson Collier et Michael North. « Parallel Agent-Based Simulation with Repast for High Performance Computing ». In : *SIMULATION* (6 nov. 2012). doi : 10.1177/0037549712462620.

[43] U. P. C. Consortium, Dan Bonachea et Gary Funck. *UPC Language and Library Specifications, Version 1.3.* LBNL-6623E. Lawrence Berkeley National Lab. (LBNL), Berkeley, CA (United States), 16 nov. 2013. doi : 10.2172/1134233.

[44] Gennaro Cordasco, Carmine Spagnuolo et Vittorio Scarano. « Toward the New Version of D-MASON : Efficiency, Effectiveness and Correctness in Parallel and Distributed Agent-Based Simulations ». In : *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Mai 2016, p. 1803-1812. doi : 10.1109/IPDPSW.2016.52.

[45] Gennaro Cordasco et al. « Bringing Together Efficiency and Effectiveness in Distributed Simulations : The Experience with D-Mason ». In : *SIMULATION* 89.10 (1er oct. 2013), p. 1236-1253. issn : 0037-5497. doi : 10.1177/0037549713489594.

[46] Massimo Cossentino et al. « ASPECS : An Agent-Oriented Software Process for Engineering Complex Systems ». In : *Autonomous Agents and Multi-Agent Systems* 20.2 (1er mars 2010), p. 260-304. issn : 1573-7454. doi : 10.1007/s10458-009-9099-4.

[47] Gabriele D'Angelo et Moreno Marzolla. « New Trends in Parallel and Distributed Simulation : From Many-Cores to Cloud Computing ». In : *Simulation Modelling Practice and Theory* 49 (1er déc. 2014), p. 320-335. issn : 1569-190X. doi : 10.1016/j.simpat.2014.06.007.

[48] Samir Das et al.
« GTW : A Time Warp System for Shared Memory Multiprocessors ».
In : *Proceedings of Winter Simulation Conference.*
Proceedings of Winter Simulation Conference. Déc. 1994, p. 1332-1339.
DOI : 10.1109/WSC.1994.717527.

[49] Christophe Deissenberg, Sander van der Hoog et Herbert Dawid.
« EURACE : A Massively Parallel Agent-Based Model of the European Economy ».
In : *Applied Mathematics and Computation* 204.2 (oct. 2008), p. 541-552.
ISSN : 00963003. DOI : 10.1016/j.amc.2008.05.116.

[50] Giani Di Caro et Marco Dorigo.
« AntNet : Distributed Stigmergetic Control for Communications Networks ».
In : *Journal of Artificial Intelligence Research* 9 (1er déc. 1998), p. 317-365.
ISSN : 1076-9757. DOI : 10.1613/jair.530.

[51] Edsger W. Dijkstra, W. H. J. Feijen et A. J. M. van Gasteren.
« Derivation of a Termination Detection Algorithm for Distributed Computations ».
In : *Control Flow and Data Flow : Concepts of Distributed Programming.*
Sous la dir. de Manfred Broy. Springer Study Edition.
Berlin, Heidelberg : Springer, 1986, p. 507-512. ISBN : 978-3-642-82921-5.
DOI : 10.1007/978-3-642-82921-5_13.

[52] Giovanni Dosi et Andrea Roventini. « Agent-Based Macroeconomics and Classical Political Economy : Some Italian Roots ».
In : *Italian Economic Journal* 3.3 (1er nov. 2017), p. 261-283. ISSN : 2199-3238.
DOI : 10.1007/s40797-017-0065-z.

[53] Joshua M. Epstein et Robert Axtell.
Growing Artificial Societies : Social Science from the Bottom Up.
Brookings Institution Press, 11 oct. 1996. 234 p. ISBN : 978-0-262-05053-1.
Google Books : xXvelSs2caQC.

[54] August Ernstsson et al. « SkePU 3 : Portable High-Level Programming of Heterogeneous Systems and HPC Clusters ». In : *International Journal of Parallel Programming* 49.6 (1er déc. 2021), p. 846-866. ISSN : 1573-7640.
DOI : 10.1007/s10766-021-00704-3.

[55] Jacques Ferber et Olivier Gutknecht. « A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems ». In : *Proceedings International Conference on Multi Agent Systems (Cat. No.98EX160).*
Proceedings International Conference on Multi Agent Systems (Cat. No.98EX160). Juill. 1998, p. 128-135. DOI : 10.1109/ICMAS.1998.699041.

[56] Jacques Ferber et Jean-Pierre Müller.
« Influences and Reaction : A Model of Situated Multiagent Systems ».
In : *Proceedings of second international conference on multi-agent systems (ICMAS-96)* (1996), p. 72-79.

[57] Stan Franklin et Art Graesser.
« Is It an Agent, or Just a Program ? : A Taxonomy for Autonomous Agents ».
In : *Intelligent Agents III Agent Theories, Architectures, and Languages*. Sous la dir. de Jörg P. Müller, Michael J. Wooldridge et Nicholas R. Jennings.
Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 1997,
p. 21-35. isbn : 978-3-540-68057-4. doi : 10.1007/BFb0013570.

[58] Richard M. Fujimoto. « Parallel and Distributed Discrete Event Simulation : Algorithms and Applications ».
In : *Proceedings of 1993 Winter Simulation Conference - (WSC '93)*.
Proceedings of 1993 Winter Simulation Conference - (WSC '93). Déc. 1993,
p. 106-114. doi : 10.1109/WSC.1993.718035.

[59] Richard M. Fujimoto. « Parallel Discrete Event Simulation ».
In : *Communications of the ACM* 33.10 (1er oct. 1990), p. 30-53.
issn : 0001-0782. doi : 10.1145/84537.84545.

[60] Michael R. Garey et David S. Johnson.
Computers and Intractability : A Guide to the Theory of NP-Completeness.
First Edition. Series of Books in the Mathematical Sciences.
W. H. Freeman, 1979. isbn : 0-7167-1045-5 978-0-7167-1045-5.

[61] Benoit Gaudou et al. « COMOKIT : A Modeling Kit to Understand, Analyze, and Compare the Impacts of Mitigation Policies Against the COVID-19 Epidemic at the Scale of a City ». In : *Frontiers in Public Health* 0 (2020).
issn : 2296-2565. doi : 10.3389/fpubh.2020.563247.

[62] Zaiyi Guo, Peter M. A. Sloot et Joc Cing Tay.
« A Hybrid Agent-Based Approach for Modeling Microbiological Systems ».
In : *Journal of Theoretical Biology* 255.2 (21 nov. 2008), p. 163-175.
issn : 0022-5193. doi : 10.1016/j.jtbi.2008.08.008.

[63] Olivier Gutknecht et Jacques Ferber.
« The MadKit Agent Platform Architecture ». In : *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*.
Sous la dir. de Tom Wagner et Omer F. Rana.
Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2001,
p. 48-55. isbn : 978-3-540-47772-3. doi : 10.1007/3-540-47772-1_5.

[64] Ross A. Hammond et Robert Axelrod. « The Evolution of Ethnocentrism ».
In : *Journal of Conflict Resolution* 50.6 (1er déc. 2006), p. 926-936.
issn : 0022-0027. doi : 10.1177/0022002706293470.

[65] Andreas Horni, Kai Nagel et Kay W. Axhausen.
The Multi-Agent Transport Simulation MATSim. Ubiquity Press, 10 août 2016.
isbn : 978-1-909188-77-8 978-1-909188-75-4 978-1-909188-78-5
978-1-909188-76-1. doi : 10.5334/baw.

[66] Andreas Huth et Christian Wissel.
« The Simulation of the Movement of Fish Schools ».
In : *Journal of Theoretical Biology* 156.3 (7 juin 1992), p. 365-385.
issn : 0022-5193. doi : 10.1016/S0022-5193(05)80681-2.

[67] David R. Jefferson. « Virtual Time ». In : *ACM Transactions on Programming Languages and Systems* 7.3 (1er juill. 1985), p. 404-425.
issn : 0164-0925. doi : 10.1145/3916.3988.

[68] Richard M. Karp. « Reducibility among Combinatorial Problems ».
In : *Complexity of Computer Computations*. Sous la dir. de
Raymond E. Miller, James W. Thatcher et Jean D. Bohlinger.
The IBM Research Symposia Series. Boston, MA : Springer US, 1972, p. 85-103.
isbn : 978-1-4684-2001-2. doi : 10.1007/978-1-4684-2001-2_9.

[69] George Karypis et Vipin Kumar.
« Multilevel K-Way Hypergraph Partitioning ».
In : *VLSI Design* 11 (1999), e19436. issn : 1065-514X.
doi : 10.1155/2000/19436.

[70] Georges Karypis et Vipin Kumar.
« Multilevel Algorithms for Multi-Constraint Graph Partitioning ».
In : *SC '98 : Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*.
SC '98 : Proceedings of the 1998 ACM/IEEE Conference on Supercomputing.
Nov. 1998, p. 28-28. doi : 10.1109/SC.1998.10018.

[71] Ayesha Kashif et al. « Simulating the Dynamics of Occupant Behaviour for Power Management in Residential Buildings ».
In : *Energy and Buildings* 56 (1er jan. 2013), p. 85-93. issn : 0378-7788.
doi : 10.1016/j.enbuild.2012.09.042.

[72] Mariam Kiran et al. « FLAME : Simulating Large Populations of Agents on Parallel Hardware Architectures ».
In : *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems : Volume 1 - Volume 1*. AAMAS '10.
Toronto, Canada : International Foundation for Autonomous Agents and Multiagent Systems, 10 mai 2010, p. 1633-1636. isbn : 978-0-9826571-1-9.

[73] Yoann Kubera, Philippe Mathieu et Sébastien Picault. « IODA : An Interaction-Oriented Approach for Multi-Agent Based Simulations ». In :
Autonomous Agents and Multi-Agent Systems 23.3 (1er nov. 2011), p. 303-343.
issn : 1573-7454. doi : 10.1007/s10458-010-9164-z.

[74] Leslie Lamport.
« Time, Clocks, and the Ordering of Events in a Distributed System ».
In : *Communications of the ACM* 21.7 (1er juill. 1978), p. 558-565.
ISSN : 0001-0782. DOI : 10.1145/359545.359563.

[75] Vincent Laperrière et al. « Structural Validation of an Individual-Based Model for Plague Epidemics Simulation ».
In : *Ecological Complexity*. Special Section : Environmental Micro-Simulation : From Data Approximation to Theory Assessment 6.2 (1er juin 2009), p. 102-112.
ISSN : 1476-945X. DOI : 10.1016/j.ecocom.2008.08.001.

[76] Guillaume Laville et al. « Using GPU for Multi-Agent Soil Simulation ».
In : *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Fév. 2013, p. 392-399.
DOI : 10.1109/PDP.2013.63.

[77] Pablo Alvarez Lopez et al. « Microscopic Traffic Simulation Using SUMO ».
In : *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. 2018 21st International Conference on Intelligent Transportation Systems (ITSC). Nov. 2018, p. 2575-2582. DOI : 10.1109/ITSC.2018.8569938.

[78] John C.S. Lui et M.F. Chan. « An Efficient Partitioning Algorithm for Distributed Virtual Environment Systems ». In : *IEEE Transactions on Parallel and Distributed Systems* 13.3 (mars 2002), p. 193-211. ISSN : 1558-2183.
DOI : 10.1109/71.993202.

[79] Sean Luke et al. « MASON : A Multiagent Simulation Environment ».
In : *SIMULATION* 81.7 (1er juill. 2005), p. 517-527. ISSN : 0037-5497.
DOI : 10.1177/0037549705058073.

[80] Nancy A. Lynch. « Shared Variable Types ». In : *Distributed Algorithms*. T. 9.4. Elsevier, 16 avr. 1996, p. 244-249. ISBN : 978-0-08-050470-4.

[81] Charles M. Macal. « Everything You Need to Know about Agent-Based Modelling and Simulation ».
In : *Journal of Simulation* 10.2 (1er mai 2016), p. 144-156. ISSN : 1747-7778.
DOI : 10.1057/jos.2016.7.

[82] Charles M. Macal et al. « CHISIM : AN AGENT-BASED SIMULATION MODEL OF SOCIAL INTERACTIONS IN A LARGE URBAN AREA ».
In : *2018 Winter Simulation Conference (WSC)*.
2018 Winter Simulation Conference (WSC). Déc. 2018, p. 810-820.
DOI : 10.1109/WSC.2018.8632409.

[83] Artur Malinowski et Paweł Czarnul. « Multi-Agent Large-Scale Parallel Crowd Simulation with NVRAM-based Distributed Cache ».
In : *Journal of Computational Science* 33 (1er avr. 2019), p. 83-94.
ISSN : 1877-7503. DOI : 10.1016/j.jocs.2019.04.004.

[84] Artur Malinowski et al.
« Multi-Agent Large-Scale Parallel Crowd Simulation ». In : *Procedia Computer Science*. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland 108 (1er jan. 2017), p. 917-926. issn : 1877-0509. doi : 10.1016/j.procs.2017.05.036.

[85] Nicolas Marilleau, Christophe Lang et Patrick Giraudoux.
« Coupling Agent-Based with Equation-Based Models to Study Spatially Explicit Megapopulation Dynamics ».
In : *Ecological Modelling* 384 (24 sept. 2018), p. 34-42.
doi : 10.1016/j.ecolmodel.2018.06.011.

[86] Nicolas Marilleau et al. « Multiscale MAS Modelling to Simulate the Soil Environment : Application to Soil Ecology ».
In : *Simulation Modelling Practice and Theory* 16.7 (1er août 2008), p. 736-745.
issn : 1569-190X. doi : 10.1016/j.simpat.2008.04.021.

[87] Dominique Masse et al. « MIOR : An Individual-Based Model for Simulating the Spatial Patterns of Soil Organic Matter Microbial Decomposition ».
In : *European Journal of Soil Science* 58.5 (2007), p. 1127-1135.
issn : 1365-2389. doi : 10.1111/j.1365-2389.2007.00900.x.

[88] Philippe Mathieu et Yann Secq. « ENVIRONMENT UPDATING AND AGENT SCHEDULING POLICIES IN AGENT-BASED SIMULATORS ».
In : *Proceedings of the 4th International Conference on Agents and Artificial Intelligence*. International Conference on Agents and Artificial Intelligence. T. 1. SciTePress, 2012, p. 170-175. isbn : 978-989-8425-96-6.
doi : 10.5220/0003732301700175.

[89] Makoto Matsumoto et Takuji Nishimura.
« Mersenne Twister : A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator ». In : *ACM Transactions on Modeling and Computer Simulation* 8.1 (1er jan. 1998), p. 3-30. issn : 1049-3301.
doi : 10.1145/272991.272995.

[90] Fabien Michel. « The IRM4S Model : The Influence/Reaction Principle for Multiagent Based Simulation ». In : *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '07. Honolulu, Hawaii : Association for Computing Machinery, 14 mai 2007, p. 1-3.
isbn : 978-81-904262-7-5. doi : 10.1145/1329125.1329289.

[91] Stanley Milgram. « The Small World Problem ».
In : *Psychology today* 2.1 (1967), p. 60-67.

[92] Jayadev Misra. « Distributed Discrete-Event Simulation ».
In : *ACM Computing Surveys* 18.1 (1er mars 1986), p. 39-65. issn : 0360-0300.
doi : 10.1145/6462.6485.

[93] Gildas Morvan et Yoann Kubera.
« On Time and Consistency in Multi-Level Agent-Based Simulations ».
7 mars 2017. doi : 10.48550/arXiv.1703.02399. arXiv : 1703.02399 [cs].

[94] Gildas Morvan, Alexandre Veremme et Daniel Dupont.
« IRM4MLS : The Influence Reaction Model for Multi-Level Simulation ».
In : *Multi-Agent-Based Simulation XI.*
Sous la dir. de Tibor Bosse, Armando Geller et Catholijn M. Jonker.
Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2011,
p. 16-27. isbn : 978-3-642-18345-4. doi : 10.1007/978-3-642-18345-4_2.

[95] MPI Forum. *MPI : A Message-Passing Interface Standard 3.1.* Rapp. tech.
4 juin 2015, p. 868.
url : https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visité
le 06/01/2022).

[96] Michael J. North et al.
« Complex Adaptive Systems Modeling with Repast Simphony ».
In : *Complex Adaptive Systems Modeling* 1.1 (13 mars 2013), p. 3.
issn : 2194-3206. doi : 10.1186/2194-3206-1-3.

[97] Robert W. Numrich et John Reid.
« Co-Array Fortran for Parallel Programming ».
In : *ACM SIGPLAN Fortran Forum* 17.2 (1er août 1998), p. 1-31.
issn : 1061-7264. doi : 10.1145/289918.289920.

[98] OpenMP Architecture Review Board. *OpenMP API Specification 5.2.*
Rapp. tech. Nov. 2021, p. 669. url : https://www.openmp.org/wp-content/up
loads/OpenMP-API-Specification-5-2.pdf (visité le 06/01/2022).

[99] Stephen K. Park et Keith W. Miller.
« Random Number Generators : Good Ones Are Hard to Find ».
In : *Communications of the ACM* 31.10 (1er oct. 1988), p. 1192-1201.
issn : 0001-0782. doi : 10.1145/63039.63042.

[100] Dirk Pawlaszczyk et Steffen Strassburger.
« Scalability in Distributed Simulations of Agent-Based Models ».
In : *Proceedings of the 2009 Winter Simulation Conference (WSC).*
Proceedings of the 2009 Winter Simulation Conference (WSC). Déc. 2009,
p. 1189-1200. doi : 10.1109/WSC.2009.5429429.

[101] François Pellegrini et Jean Roman.
« Scotch : A Software Package for Static Mapping by Dual Recursive
Bipartitioning of Process and Architecture Graphs ».
In : *High-Performance Computing and Networking.*
Sous la dir. d'Heather Liddell et al. Lecture Notes in Computer Science.
Berlin, Heidelberg : Springer, 1996, p. 493-498. isbn : 978-3-540-49955-8.
doi : 10.1007/3-540-61142-8_588.

[102] Manisa Pipattanasomporn, Hasan Feroze et Saifur Rahman. « Multi-Agent Systems in a Distributed Smart Grid : Design and Implementation ».
In : *2009 IEEE/PES Power Systems Conference and Exposition*.
2009 IEEE/PES Power Systems Conference and Exposition. Mars 2009, p. 1-8.
DOI : 10.1109/PSCE.2009.4840087.

[103] Konstantin Popov et al.
« Parallel Agent-Based Simulation on a Cluster of Workstations ».
In : *Parallel Processing Letters* 13.04 (1er déc. 2003), p. 629-641.
ISSN : 0129-6264. DOI : 10.1142/S0129626403001562.

[104] Saifur Rahman, Manisa Pipattanasomporn et Yonael Teklu.
« Intelligent Distributed Autonomous Power Systems (IDAPS) ».
In : *2007 IEEE Power Engineering Society General Meeting*.
2007 IEEE Power Engineering Society General Meeting. Juin 2007, p. 1-8.
DOI : 10.1109/PES.2007.386043.

[105] Dhananjai M. Rao.
« Accelerating Parallel Agent-Based Epidemiological Simulations ».
In : *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM PADS '14.
New York, NY, USA : Association for Computing Machinery, 18 mai 2014,
p. 127-138. ISBN : 978-1-4503-2794-7. DOI : 10.1145/2601381.2601387.

[106] Dhananjai M. Rao et Alexander Chernyakhovsky.
« Parallel Simulation of the Global Epidemiology of Avian Influenza ».
In : *2008 Winter Simulation Conference*. 2008 Winter Simulation Conference.
Déc. 2008, p. 1583-1591. DOI : 10.1109/WSC.2008.4736241.

[107] Paul Richmond. « Resolving Conflicts between Multiple Competing Agents in Parallel Simulations ». In : *Euro-Par 2014 : Parallel Processing Workshops*.
Sous la dir. de Luís Lopes et al. Lecture Notes in Computer Science.
Cham : Springer International Publishing, 2014, p. 383-394.
ISBN : 978-3-319-14325-5. DOI : 10.1007/978-3-319-14325-5_33.

[108] Paul Richmond et Mozhgan K. Chimeh.
« FLAME GPU : Complex System Simulation Framework ». In : *2017 International Conference on High Performance Computing Simulation (HPCS)*.
2017 International Conference on High Performance Computing Simulation (HPCS). Juill. 2017, p. 11-17. DOI : 10.1109/HPCS.2017.12.

[109] Omar Rihawi, Yann Secq et Philippe Mathieu.
« Synchronization Policies Impact in Distributed Agent-Based Simulation ».
In : *Distributed Computing and Artificial Intelligence*.
Sous la dir. de Sigeru Omatu et al.
Advances in Intelligent Systems and Computing.
Cham : Springer International Publishing, 2013, p. 19-26.
ISBN : 978-3-319-00551-5. DOI : 10.1007/978-3-319-00551-5_3.

[110] Sebastian Rodriguez, Nicolas Gaud et Stéphane Galland.
« SARL : A General-Purpose Agent-Oriented Programming Language ».
In : *2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*.
2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). T. 3. Août 2014, p. 103-110.
DOI : 10.1109/WI-IAT.2014.156.

[111] Alban Rousset. « Contribution à la distribution et à la synchronisation des Systèmes Multi-Agents sur les super-calculateurs ».
These de doctorat. Besançon, 14 oct. 2016.
URL : http://theses.fr/2016BESA2043 (visité le 28/04/2022).

[112] Alban Rousset et al. « A Survey on Parallel and Distributed Multi-Agent Systems for High Performance Computing Simulations ».
In : *Computer Science Review* 22 (1er nov. 2016), p. 27-46. ISSN : 1574-0137.
DOI : 10.1016/j.cosrev.2016.08.001.

[113] Alban Rousset et al. « Using Nested Graphs to Distribute Parallel and Distributed Multi-agent Systems ». In : *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*.
2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). Fév. 2016, p. 710-717.
DOI : 10.1109/PDP.2016.91.

[114] Xavier Rubio-Campillo.
« Pandora : A Versatile Agent-Based Modelling Platform for Social Simulation ».
In : *Proceedings of SIMUL*. SIMUL 2014 : The Sixth International Conference on Advances in System Simulation. 1er jan. 2014, p. 29-34.
DOI : 10.13140/2.1.5149.4086.

[115] Stuart J. Russell et Peter Norvig.
Artificial Intelligence : A Modern Approach. Fourth edition.
Pearson Series in Artificial Intelligence. Hoboken : Pearson, 2021.
ISBN : 978-0-13-461099-3.

[116] David Scerri et al. « An Architecture for Modular Distributed Simulation with Agent-Based Models ». In : *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*. AAMAS 2010. T. 1.
Toronto, Canada, 2010, p. 541-548.

[117] Thomas C. Schelling. « Dynamic Models of Segregation ».
In : *The Journal of Mathematical Sociology* 1.2 (1er juill. 1971), p. 143-186.
ISSN : 0022-250X. DOI : 10.1080/0022250X.1971.9989794.

[118] Kirk Schloegel, George Karypis et Vipin Kumar.
« Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning ».
In : *Euro-Par 2000 Parallel Processing*. Sous la dir. d'Arndt Bode et al.

Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2000, p. 296-310. ISBN : 978-3-540-44520-3. DOI : 10.1007/3-540-44520-X_39.

[119] Lisa M. SOKOL, Jon B. WEISSMAN et Paula A. MUTCHLER. *MTW : An Empirical Performance Study.* Institute of Electrical and Electronics Engineers (IEEE), 1991. URL : https://repository.lib.ncsu.edu/bitstream/handle/1840.4/4538/1991_0076.pdf?sequence=1 (visité le 08/02/2022).

[120] Roberto SOLAR, Remo SUPPI et Emilio LUQUE. « High Performance Distributed Cluster-Based Individual-Oriented Fish School Simulation ». In : *Procedia Computer Science*. Proceedings of the International Conference on Computational Science, ICCS 2011 4 (1er jan. 2011), p. 76-85. ISSN : 1877-0509. DOI : 10.1016/j.procs.2011.04.009.

[121] Roberto SOLAR, Remo SUPPI et Emilio LUQUE. « Proximity Load Balancing for Distributed Cluster-based Individual-oriented Fish School Simulations ». In : *Procedia Computer Science*. Proceedings of the International Conference on Computational Science, ICCS 2012 9 (1er jan. 2012), p. 328-337. ISSN : 1877-0509. DOI : 10.1016/j.procs.2012.04.035.

[122] Vinoth SURYANARAYANAN, Georgios THEODOROPOULOS et Michael LEES. « PDES-MAS : Distributed Simulation of Multi-agent Systems ». In : *Procedia Computer Science*. 2013 International Conference on Computational Science 18 (1er jan. 2013), p. 671-681. ISSN : 1877-0509. DOI : 10.1016/j.procs.2013.05.231.

[123] Patrick TAILLANDIER et al. « Building, Composing and Experimenting Complex Spatial Models with the GAMA Platform ». In : *GeoInformatica* 23.2 (1er avr. 2019), p. 299-322. ISSN : 1573-7624. DOI : 10.1007/s10707-018-00339-6.

[124] Georgios THEODOROPOULOS et Brian LOGAN. « A FRAMEWORK FOR THE DISTRIBUTED SIMULATION OF AGENT-BASED SYSTEMS ». In : *Modelling and Simulation : A Tool for the next Millenium, Proceedings of the 13th European Simulation Multiconference (ESM'99)*. T. 1. 1999, p. 58-65.

[125] Guillermo VIGUERAS et al. « A Comparative Study of Partitioning Methods for Crowd Simulations ». In : *Applied Soft Computing* 10.1 (1er jan. 2010), p. 225-235. ISSN : 1568-4946. DOI : 10.1016/j.asoc.2009.07.004.

[126] Guillermo VIGUERAS et al. « A Scalable Multiagent System Architecture for Interactive Applications ». In : *Science of Computer Programming*. Special Section : The Programming Languages Track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special Section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments

78.6 (1er juin 2013), p. 715-724. ISSN : 0167-6423.
DOI : 10.1016/j.scico.2011.09.002.

[127] Haoliang WANG et al.
« Scalability in the MASON Multi-Agent Simulation System ».
In : *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT). Oct. 2018, p. 1-10. DOI : 10.1109/DISTRA.2018.8601006.

[128] Duncan J. WATTS et Steven H. STROGATZ.
« Collective Dynamics of 'Small-World' Networks ».
In : *Nature* 393.6684 (6684 juin 1998), p. 440-442. ISSN : 1476-4687.
DOI : 10.1038/30918.

[129] Danny WEYNS, Alexander HELLEBOOGH et Tom HOLVOET. « The Packet-World : A Test Bed for Investigating Situated Multi-Agent Systems ».
In : *Software Agent-Based Applications, Platforms and Development Kits*.
Sous la dir. de Rainer UNLAND, Monique CALISTI et Matthias KLUSCH.
Whitestein Series in Software Agent Technologies. Basel : Birkhäuser, 2005, p. 383-408. ISBN : 978-3-7643-7348-1. DOI : 10.1007/3-7643-7348-2_16.

[130] Uri WILENSKY et Kenneth REISMAN. « Thinking Like a Wolf, a Sheep, or a Firefly : Learning Biology Through Constructing and Testing Computational Theories—An Embodied Modeling Approach ».
In : *Cognition and Instruction* 24.2 (1er juin 2006), p. 171-209. ISSN : 0737-0008.
DOI : 10.1207/s1532690xci2402_1.

[131] Yadong XU et al.
« Relaxing Synchronization in Parallel Agent-Based Road Traffic Simulation ».
In : *ACM Transactions on Modeling and Computer Simulation* 27.2 (27 mai 2017). ISSN : 1049-3301. DOI : 10.1145/2994143.

[132] Huan ZHOU et al.
« DART-MPI : An MPI-based Implementation of a PGAS Runtime System ».
In : *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14.
New York, NY, USA : Association for Computing Machinery, 6 oct. 2014, p. 1-11. ISBN : 978-1-4503-3247-7. DOI : 10.1145/2676870.2676875.

Annexes

Annexe A.

Sérialisation *ObjectPack*

Le principe de base de la sérialisation *ObjectPack* a déjà été introduit dans la section 3.5.1. Nous fournissons ici quelques détails supplémentaires relatifs à l'implémentation de cette technique de sérialisation.

L'algorithme 19 présente le point d'entrée de la sérialisation/déserialisation avec la librairie *ObjectPack*. L'espace mémoire est alloué automatiquement par la méthode Sérialiser, de manière transparente à l'utilisateur, qui peut avoir accès aux données sérialisées à envoyer grâce à l'espace mémoire référencé par la méthode EspaceMémoire. Les données reçues peuvent être stockées directement dans l'espace mémoire grâce à cette même référence, puis désérialisées grâce à la méthode Désérialiser. Les appels aux méthodes TailleMémoire, Écrire et Lire font implicitement référence aux règles de sérialisation définies pour le type de l'élément passé en paramètre, qu'elles soient prédéfinies par la librairie ou définies par l'utilisateur. La position actuelle fait référence à la position à laquelle lire ou écrire les données dans l'espace mémoire, utilisée notamment pour la sérialisation des types fondamentaux.

Algorithme 19 Sérialisation ObjectPack

```
1: algorithme Sérialiser(élément)
2:     Allouer un espace mémoire de taille TailleMémoire(élément)
3:     Initialiser la position actuelle au début de l'espace mémoire
4:     Écrire(espace mémoire, élément)
5: fin algorithme

6: algorithme Désérialiser(élément)
7:     Initialiser l'élément à une valeur par défaut          ▷ allocation de mémoire
8:     Initialiser la position actuelle au début de l'espace mémoire
9:     Lire(espace mémoire, élément)
10: fin algorithme

11: algorithme EspaceMémoire
12:     retourner référence vers l'espace mémoire
13: fin algorithme
```

Un exemple de sérialisation pour un type fondamental est donné dans l'algorithme 20. La position actuelle est incrémentée au fur et à mesure des écritures ou des lectures dans

l'espace. Dans les langages C ou C++, la copie des données est typiquement réalisée directement grâce à la méthode `memcpy`, d'où l'efficacité de la méthode.

Algorithme 20 Exemple de sérialisation d'un type fondamental (entier)

```
1: algorithme TailleMémoire(entier)
2:     retourner taille de l'entier en octets
3: fin algorithme

4: algorithme Écrire(espace mémoire, entier)
5:     Copier l'entier dans l'espace mémoire à la position actuelle
6:     Avancer la position actuelle de TailleMémoire(entier)
7: fin algorithme

8: algorithme Lire(espace mémoire, entier)
9:     Copier les données de l'espace mémoire à la position actuelle dans l'entier
10:    Avancer la position actuelle de TailleMémoire(entier)
11: fin algorithme
```

Un exemple de fonctions de sérialisation pour un tableau dynamique est proposé dans l'algorithme 21. Le tableau contient des éléments de type arbitraire. Le tableau est donc lui même un type dit composé, et non fondamental.

Les appels aux méthodes TailleMémoire (ligne 4), Écrire (ligne 11) et Lire (ligne 18) correspondent à des appels récursifs aux méthodes de sérialisation associées au type d'élément du tableau. La récursion s'arrête lorsqu'un type fondamental est atteint. Pour la plupart des types composés, l'utilisateur appelle seulement les fonctions de sérialisation prédéfinies par la librairie, sans avoir à se soucier de la gestion de l'espace mémoire interne à la librairie, d'où un haut niveau d'abstraction.

Algorithme 21 Exemple de sérialisation d'un tableau dynamique

```
1: algorithme TailleMémoire(tableau dynamique)
2:     taille ← TailleMémoire(nombre d'éléments du tableau)
3:     pour chaque élément du tableau faire
4:         taille ← taille + TailleMémoire(élément)
5:     fin pour
6:     retourner taille
7: fin algorithme

8: algorithme Écrire(espace mémoire, tableau dynamique)
9:     Écrire(espace mémoire, taille du tableau)
10:    pour chaque élément du tableau faire
11:        Écrire(espace mémoire, élément)
12:    fin pour
13: fin algorithme

14: algorithme Lire(espace mémoire, tableau dynamique)
15:    Lire(espace mémoire, nombre d'éléments du tableau)
16:    Capacité du tableau dynamique ← nombre d'éléments
17:    pour i de 0 à nombre d'éléments faire
18:        Lire(espace mémoire, tableau[i])
19:    fin pour
20: fin algorithme
```

Annexe B.

Discussion sur les coûts de communication du *Méta-Modèle*

La modélisation des communications dans le *Méta-modèle* soulève quelques questions. Dans un modèle réel, le coût réel des communications entre deux agents dépend de nombreux paramètres, tels que le comportement des agents, le volume des données à transmettre, ainsi que le mode de synchronisation des données qui définit les modalités d'interactions avec un agent *distant*. Ces modes sont détaillés dans le chapitre 5. Néanmoins nous apportons ici quelques éléments de réflexion génériques sur la gestion des communications.

On peut distinguer deux cas.

— Le coût réel de communication $\gamma_{\mathcal{S}}(a_i, a_j)$ est constant, quel que soit le comportement de a_i. C'est par exemple le cas quand la synchronisation consiste à importer à la fin de chaque pas de temps les données à jours de a_j, comme dans le cas de D-MASON, de RepastHPC, ou du `GhostMode` de FPMAS. En effet, une fois l'agent importé, le comportement de a_i n'engendrera pas de nouvelle communication. L'importation de plusieurs agents peut alors être réalisée grâce à des communications dites collectives. Le coût des communications lié à l'importation de a_j ne s'applique qu'une seule fois, indépendamment du nombre d'interactions de a_i avec a_j, et indépendamment du nombres d'agents *locaux* interagissant avec a_j. En effet, si deux agents *locaux* sont amenés à interagir avec un même agent *distant*, le coût réel de communication est mutualisé entre les deux agents *locaux* car les données de l'agent *distant* ne sont importés qu'une fois pour les deux.

— Le coût réel de la communication $\gamma_{\mathcal{S}}(a_i, a_j)$ dépend du comportement de a_i. C'est notamment le cas avec les méthodes qui consistent à importer à la volée les données de l'agent a_j lorsqu'une interaction est nécessaire, grâce à des communications point à point, comme avec le `HardSyncMode` de FPMAS. Dans ce cas, le coût de l'importation de a_j peut s'appliquer plusieurs fois si plusieurs interactions avec a_j ont lieu pendant l'exécution du comportement de a_i. A l'inverse, aucune communication n'aura lieu si a_i n'interagit finalement pas avec a_j. Contrairement au cas précédent, le coût des communications entre a_i et a_j est alors à considérer de manière individuelle.

La méthode d'estimation du temps d'exécution présentée au chapitre 4 utilise des temps de communication constants. On compte le nombre de relations de chaque agent

local avec des agents *distants*, et on suppose que le temps de communication s'applique une fois à chaque relation. Le coût de communication simulé ne prend donc pas en compte le comportement des agents. Ce biais peut cependant se compenser en adaptant le paramètre représentant le temps de communication en fonction du nombre moyen d'interactions entre deux agents. Le coût de communication simulé s'applique ensuite de manière individuelle : on ne considère pas la mutualisation possible des coûts de communication. Le temps d'exécution estimé ne correspond donc rigoureusement à aucun des deux cas précédents. Il possède cependant l'avantage d'être indépendant du mode de synchronisation, et d'être très proche des liens réels entre agents distants. En effet, les temps de communications estimés sont directement proportionnels au nombre de liens distants. Ce modèle de communication correspond en outre exactement au calcul des coûts de communication effectué par Zoltan dans notre configuration de l'équilibrage de charge à base de graphe.

Annexe C.

Conversion d'un modèle SMA vers le modèle de graphe de Zoltan

La construction des liens et le calcul de leurs poids décrits dans l'algorithme 8 nécessitent un mécanisme permettant de convertir notre représentation de système Multi-Agents en un graphe interprétable par Zoltan. Un exemple est donné sur la figure C.1. La figure C.1a représente les interactions entre agents sous forme d'un graphe orienté, conformément à la modélisation introduite au chapitre 3 : un lien orienté de a_i vers a_j signifie qu'a_j appartient au voisinage de a_i, et donc qu'a_i peut interagir avec a_j. Un poids est associé à chaque lien. La figure C.1b montre la conversion vers un graphe non orienté exploitable par Zoltan, conformément à l'algorithme 8. On remarque que dans cette configuration, le coût en communications des partitions considéré par Zoltan est calculé de manière analogue au *Méta-modèle*, le poids des liens correspondant ici aux paramètres représentant les temps de communication. On remarque notamment qu'avec le modèle de la figure C.1b, Zoltan ne considère pas les possibles mutualisations des coûts de communications entre les agents a_1 et a_2. En effet le coût en communication du partitionnement représenté, qui associe a_1 et a_2 à un processus et a_3 à un autre, correspond à la somme des coûts individuels $w_4 + w_2 + w_3$. Le mécanisme de conversion est donc plus proche du cas où les communications dépendent du comportement des agents, le poids des liens étant défini individuellement pour chaque relation.

La figure C.2 montre un mécanisme de conversion plus adapté au cas où les communications réelles ne dépendent pas du comportement des agents, même si cette solution n'a pas été mise en pratique. On suppose tout d'abord que le poids des liens, et donc le coût des relations, ne dépend pas du comportement des agents mais des agents

(a) Modèle Multi-Agents (b) Modèle Zoltan à base de graphe non orienté

FIGURE C.1. – Transformation d'un modèle Multi-Agents vers le modèle de graphe utilisé par Zoltan.

(a) Modèle Multi-Agents (b) Modèle Zoltan à base d'hypergraphe

FIGURE C.2. – Transformation d'un modèle Multi-Agents vers le modèle d'hypergraphe utilisé par Zoltan.

avec lesquels ils sont en relation, dont les données sont importées à chaque itération. Ainsi les coûts des relations de a_1 à a_3 et de a_2 à a_3 sont égaux et valent w_3. Le coût du partitionnement considéré avec la méthode de la figure C.1b, d'une valeur de $w_3+w_3+w_2$, est alors surestimé car a_3 n'est importé qu'une seule fois donc le coût w_3 n'est réellement appliqué qu'une fois. Le modèle à base d'hypergraphe de la figure C.2b permet de prendre en compte la mutualisation des coûts de communication, en assignant un coût de w_3+w_2 au partitionnement représenté, plus réaliste dans le cas où le coût réel des communications ne dépend pas du comportement des agents.

Ainsi l'adaptation en graphe simple représente mieux les coûts réels du `HardSyncMode` de FPMAS, et l'adaptation en hypergraphe représente mieux ceux du `GhostMode` ou du `GlobalGhostMode`. Même si ces considérations permettent d'adapter finement Zoltan au cas de la simulation distribuée de SMA, nos expérimentations avec la conversion à base de graphe simple ainsi que des poids unitaires permettent déjà d'obtenir des résultats robustes, l'objectif de Zoltan consistant dans tous les cas à minimiser la quantité totale de communications.

Titre : Distribution et synchronisation des simulations de Systèmes Multi-Agents

Mots clés : Systèmes Multi-Agents, Simulation Distribuée, Équilibrage de Charge, Synchronisation des Données

Résumé : La simulation de Systèmes Multi-Agents (SMA) permet d'expliquer et de prédire le comportement des systèmes complexes dans de nombreux domaines tels que l'épidémiologie, l'économie ou l'environnement. La grande taille des modèles étudiés mène à l'utilisation du Calcul Haute Performance et de la simulation distribuée pour lever ces limites. L'aspect naturellement parallèle des agents en fait d'excellents candidats à l'exécution distribuée, qui pose cependant de nombreux problèmes, comme la continuité des données, l'équilibrage de charge ou la synchronisation des données entre les processus. Une architecture logicielle générique permettant de résoudre ces problèmes de manière flexible et indépendamment du contexte de développement est proposée. Une conception par interface fait émerger des composants indépendants et abstraits nécessaires à la distribution de toute simulation de SMA. Nous proposons une analyse qualitative et quantitative de méthodes d'équilibrage de charge d'une part, notamment basées sur l'application de partitionnements de graphe à la simulation distribuée de SMA, et de modes de synchronisation des données d'autre part, dont certains permettent la gestion des lectures et écritures concurrentes entre les processus. Cette analyse montre que les avantages de chaque méthode dépendent des modèles et des besoins des utilisateurs, d'où l'intérêt de la conception de plateformes de simulation modulables basées sur des interfaces permettant de facilement intégrer de nouvelles méthodes. L'architecture logicielle proposée est essentiellement issue de notre expérience de développement de FPMAS, une plateforme C++ de simulation distribuée de SMA basée sur les solutions proposées.

Title : Distribution and synchronisation of Multi-Agent Systems

Keywords : Multi-Agent Systems, Distributed Simulation, Load Balancing, Data Synchronisation

Abstract : Multi-Agent Systems (MAS) simulation allows to explain and predict the behavior of complex systems in various fields such as epidemiology, economy or environment. The large size of studied models leads to the usage of High Performance Computing and distributed simulation to overcome those limits. The naturally parallel aspect of agents make them excellent candidates to the distributed execution, that however poses many problems, such as data continuity, load balancing or data synchronisation between processes. A generic software architecture that allows to solve those problems in a flexible way and independently from the development context is proposed. An interface based design brings out independent and abstract components required for the distribution of any MAS simulation. We propose a qualitative and quantitative analysis of load balancing methods on the one hand, notably based on the application of graph partitioning to the distributed simulation of MAS, and of data synchronisation modes on the other hand, among which some allow the management of concurrent reads and writes between processes. This analysis shows that advantages of each method depend on models or user needs, hence the interest for the design of modular simulation platforms based on interfaces that allow to easily integrate new methods. The proposed software architecture essentially comes from our development experience of FPMAS, a C++ distributed MAS simulation platform based on proposed solutions.

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ
Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon