Paul Breugnot, Bénédicte Herrmann, Christophe Lang,
Laurent Philippe, Alban Rousset

Politiques de synchronisation dans les systèmes multi-agents distribués parallèles

Volume 3, nº 5-6 (2022), p. 527-556.

DOI not yet assigned

La Revue Ouverte d'Intelligence Artificielle est membre du
Centre Mersenne pour l'édition scientifique ouverte
www.centre-mersenne.org
e-ISSN : pending

Revue Ouverte d'Intelligence Artificielle
Volume 3, n° 5-6, 2022, 527-556

Politiques de synchronisation dans les systèmes multi-agents distribués parallèles

Paul Breugnot[a], Bénédicte Herrmann[a], Christophe Lang[a], Laurent Philippe[a], Alban Rousset[b]

[a] Institut Femto-ST, Université de Bourgogne-Franche-Comté/CNRS, France
E-mail : paul.breugnot@univ-fcomte.fr, benedicte.herrmann@univ-fcomte.fr, christophe.lang@univ-fcomte.fr, laurent.philippe@univ-fcomte.fr

[b] LuxProvide S.A.,Luxembourg
E-mail : alban.rousset@lxp.lu.

RÉSUMÉ. — Parmi les méthodes de modélisation/simulation, les systèmes multi-agents présentent un intérêt particulier pour simuler les systèmes complexes. Lorsque la taille des modèles croît, le recours à la simulation distribuée est nécessaire mais pose de nombreux problèmes. Dans cet article, nous nous intéressons à l'impact de la synchronisation sur l'implémentation des modèles et leur exécution. Nous mettons en évidence des problématiques de synchronisation à travers des instances de modèles et nous analysons expérimentalement l'impact des politiques de synchronisation sur des exécutions de grande taille. En réponse aux manques mis en évidence, nous proposons une interface de synchronisation générique et son implémentation dans la plateforme de simulation FPMAS.

MOTS-CLÉS. — multi-agent simulation, parallélisme, MAS, High Performance Computing, synchronisation.

1. INTRODUCTION

La simulation numérique est devenue le troisième pilier de la science en tant qu'étape de validation de la théorie, déterminante pour le passage à l'expérimentation. Elle vise à virtualiser le monde réel, à en reproduire les comportements, par exemple pour explorer son évolution dans différentes configurations ou pour comprendre comment le contrôler. Dans les systèmes complexes, plusieurs phénomènes peuvent ainsi être étudiés simultanément mais les comportements sont souvent trop élaborés et interdépendants pour pouvoir être modélisés par une loi unique. Les systèmes multi-agents sont alors souvent utilisés pour modéliser les comportements dynamiques des entités qui composent le système car ils reposent sur une description algorithmique simple d'agents qui interagissent entre eux. De nombreuses plateformes [11, 23, 24] proposent un environnement de développement pour de tels modèles.

La qualité d'une simulation dépend bien souvent de la taille et de la précision du modèle. Or l'accroissement de la taille du modèle et de sa précision entraîne, de fait,

une augmentation du nombre de calculs réalisés et rend nécessaire le recours à des exécutions parallèles, voire à l'utilisation de moyens de calcul haute performance (HPC : High Performance Computing). Si la simulation sur un seul ordinateur est souvent complexe, l'exécution distribuée parallèle d'une simulation est un vrai enjeu car elle pose de nombreux problèmes comme la distribution de l'environnement, la communication entre les instances parallèles de la plateforme, etc. Il existe des plateformes multi-agents (Parallel and Distributed MAS ou PDMAS) qui prennent en charge tout ou partie de l'exécution distribuée. Plusieurs instances, ou processus, de la plateforme coopèrent pour faciliter la mise en œuvre d'un modèle sur un ensemble d'ordinateurs ou au sein d'un cluster. Parmi les solutions proposées nous pouvons distinguer plusieurs approches architecturales de découpage des fonctions de la plateforme entre les processus. Dans certaines approches toutes les machines ne jouent pas un rôle symétrique, par exemple pour les architectures hétérogènes où chaque machine peut jouer un rôle différent dans la mise en place de la simulation [25]. Pour les modèles de grande taille, les méthodes de distribution homogènes et décentralisées sont en général mieux adaptées et permettent l'exploitation efficace de machines de type cluster de calcul. Ce type de solution est souvent caractérisé par l'utilisation de la librairie MPI.

Dans ce contexte, la synchronisation des données reste l'un des points clefs pour l'exécution efficace d'une simulation multi-agents parallèle du fait des nombreux échanges et dépendances temporelles qu'elle induit. Nous nous intéressons donc, dans cet article, aux problèmes posés par la synchronisation au sein de simulations multi-agents distribuées parallèles, en visant plus particulièrement les exécutions à large échelle. Comme les plateformes multi-agents fonctionnent fréquemment par pas de temps, il est nécessaire de s'interroger sur la manière d'échanger les données entre les processus distribués d'une même simulation au regard de ce mode de fonctionnement.

Les contributions proposées dans cet article sont, d'une part, la définition de politiques – ou modes – de synchronisation qui peuvent être utilisées au sein de simulations multi-agents parallèles et leur mise en évidence au travers d'exemples de modèles et, d'autre part, l'analyse expérimentale de l'impact des modes de synchronisation sur des exécutions de grande taille, jusqu'à 512 cœurs, dans les systèmes multi-agents parallèles et distribués. Sur la base de cette analyse et des manques mis en évidence, nous avons défini une interface générique des modes de synchronisation, rendant possible l'implémentation de divers modes dans la plateforme FPMAS ainsi que leur application à n'importe quel modèle.

L'article est organisé comme suit. Dans la Section 2 nous présentons un état de l'art sur les PDMAS et leur synchronisation. Nous proposons une étude des problématiques de synchronisation des systèmes multi-agents distribués dans la Section 3. Puis, dans la Section 4, nous présentons une étude expérimentale de l'impact de différents modes de synchronisation sur l'exécution de trois modèles agents. Enfin, la définition d'une interface générique permettant l'implémentation de divers modes de synchronisation est proposée dans la Section 5.

2. PDMAS et synchronisation

Par nature les agents interagissent entre eux, soit directement en consultant les données d'autres agents, soit indirectement, à travers les modifications réalisées sur l'environnement ou par échange de messages. Dans un PDMAS, les agents du modèle sont distribués entre les instances de la plateforme, ou processus, qui prennent en charge leur animation. De la même manière, les données de l'environnement sont distribuées entre les processus. Un agent peut donc avoir à interagir avec des agents ou utiliser des données de l'environnement situées sur un autre processus. Ces accès nécessitent alors la mise en place d'une synchronisation entre les processus pour maintenir un état cohérent et permettre aux agents d'accéder à la valeur à jour des données avec l'objectif que la simulation distribuée donne le même résultat que la simulation séquentielle.

À noter que nous nous intéressons dans cet article à la simulation de systèmes multi-agents sur une architecture distribuée. Les processus étant distribués, la synchronisation des données est réalisée sous la forme de communications. L'accès à ces données est alors plus coûteux qu'un simple accès local et il est important d'implémenter la synchronisation au plus juste, de manière à limiter les surcoûts. Cette implémentation suppose de caractériser les propriétés attendues. Dans la littérature, la caractérisation de la synchronisation de modèles parallèles repose sur le mode de dépendance de données et le respect de la causalité, donc de la synchronisation temporelle.

2.1. Modes de dépendance de données

On distingue deux modes de gestion de la dépendance de données dans les simulations numériques : (i) le mode *ghost* qui utilise les données calculées au pas de temps précédent (le *ghost*) et (ii) le mode *non ghost* qui n'utilise qu'une seule instance des données, dans laquelle les résultats des calculs sont directement reportés. Le *ghost* n'est accessible qu'en lecture alors que, sans *ghost*, les données sont accessibles en lecture et en écriture et l'ordre d'accès aux données impacte les résultats [14]. Dans le cas des PDMAS, le mode *ghost* permet de ne diffuser les données modifiées qu'à la fin des pas de temps. À noter que, pour limiter le coût de ces mises à jour, plusieurs PDMAS utilisent des zones de recouvrement, copies locales des données distantes limitées à la zone de perception des agents.

Le mode *non ghost* nécessite une gestion de la concurrence des lectures et écritures réalisés par les processus parallèles. Bien que la résolution de ce problème soit relativement triviale en mémoire partagée grâce à l'utilisation de mécanismes de verrouillage, l'extension de ces solutions aux architectures distribuées pose problème. La librairie BCL [2] est un exemple d'implémentation de pointeurs distants et de structures de données distribuées. La gestion de l'accès concurrent en lecture/écriture aux éléments des structures n'est cependant pas abordée. De plus, ces travaux à eux seuls ne permettent pas de résoudre tous les problèmes inhérents à l'exécution distribuée des simulations de SMA (schéma d'exécution, équilibrage de charge, continuité des données, etc).

Dans [13], les auteurs proposent de gérer la concurrence d'accès aux cellules de l'environnement via un fichier partagé par tous les processus de la simulation.

L'attribution des cellules à chaque agent se fait cependant de manière centralisée et la concurrence d'accès entre les agents n'est pas abordée. De plus les nombreuses opérations d'écriture dans un fichier, plus coûteuses que des accès mémoire directs, posent des problèmes de passage à l'échelle. D'autres travaux se basent également sur la gestion centralisée des conflits [16, 21], 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. Il est cependant intéressant de noter que ce type d'interaction est compatible avec la méthode générique de modélisation par *Influence/Réaction* [10].

Nous analysons en 3.1 l'incidence du mode de dépendance de données sur la synchronisation dans les modèles multi-agents et nous montrons que le choix du mode est déterminé par le modèle.

2.2. Synchronisation temporelle

Deux approches de synchronisation sont définies dans le domaine des systèmes à évènements discrets : l'approche conservative et l'approche optimiste. Dans l'approche conservative (ou pessimiste), lorsqu'un processus traite un événement de date T, pour respecter la causalité, il doit être sûr qu'aucun évènement avec une date $T' < T$ ne pourra être reçu ultérieurement. On doit donc s'assurer que tous les processus sont au pas de temps T pour commencer à traiter les évènements. Les premiers algorithmes utilisant l'approche conservative ont été proposés par Chandy et al. [3]. Cette approche limite l'exploitation du parallélisme d'un modèle : en effet, l'avancée globale de la simulation est limitée par le processus le plus lent.

Contrairement à l'approche conservative, l'approche optimiste permet aux processus d'avancer indépendamment : chaque processus traite les événements dont il a connaissance sans attendre les autres processus, ce qui permet en principe de maximiser l'utilisation des ressources de calcul. Cette connaissance des événements à traiter étant locale, et donc partielle, elle peut impliquer l'omission de certains événements provenant d'autres processus et donc ne pas respecter la causalité. Lorsque le processus reçoit une donnée dont la date T' est antérieure à la date locale T, un retour en arrière est effectué par des mécanismes de *Rollback* [12] qui impliquent de sauvegarder plusieurs points de récupération par processus. Ainsi Xu et al. [29] définissent le *lookahead* comme étant la durée jusqu'au prochain pas de temps auquel il faudra mettre à jour les données. Ceci laisse la possibilité de continuer un processus tant qu'il n'est pas arrivé à son *lookahead*. Ànoter que cette approche est difficilement généralisable à tous les modèles et qu'elle n'est pas efficace lorsque le temps d'exécution passé à effectuer des *Rollback* devient très important, un *Rollback* pouvant entraîner une réaction en chaîne de *Rollback*.

Les tentatives de mise en place de la synchronisation optimiste dans le cas de la simulation distribuée de systèmes multi-agents sont rares et complexes [15, 17, 22]. En effet, cette méthode semble difficilement justifiable dans le cas des SMA où les agents interagissent classiquement de manière intensive et régulière avec les autres, augmentant grandement les probabilités de *Rollback*. De plus, la maintenance d'un

historique des transactions pour permettre les *Rollback* peut rapidement engendrer un coût en mémoire démesuré, compte tenu du nombre d'agents à simuler.

Nous considérons en particulier des PDMAS où les événements sont planifiés tous les pas de temps. L'utilisation des approches conservatives ou optimistes peut alors s'entendre comme le respect strict ou non de la frontière du pas de temps : cette étude est limitée au cas conservatif, comme pour la plupart des plateformes existantes.

2.3. La synchronisation des données dans les PDMAS

Dans [18] nous avons proposé une étude des plateformes multi-agents parallèles et distribuées. Parmi les plateformes que nous avons évaluées, seules quatre d'entre elles permettent d'envisager une exécution sur des ressources de grande taille, de type HPC.

D-Mason. — La plateforme D-Mason [7] implémente des mécanismes de synchronisation conservatifs et le mode *ghost* de gestion de dépendance des données. Pour réaliser une synchronisation conservative, chaque pas de temps est divisé en deux étapes : (1) la communication et synchronisation et (2) l'exécution de la simulation. Il y a donc une barrière de synchronisation à chaque pas de temps. Les agents d'une cellule (partition) c ne peuvent pas exécuter le pas de temps i tant que les cellules voisines n'ont pas terminé d'exécuter le pas de temps $i-1$. Àla fin d'un pas de temps, chaque cellule envoie aux cellules voisines les informations concernant les agents qui se situent dans la zone de recouvrement ou les agents qui doivent être migrés d'un processus à un autre. Pour le pas de temps i les comportements des agents de la cellule c sont ainsi calculés à partir du *ghost* des cellules voisines.

RepastHPC. — Pour gérer le partage de données, la plateforme RepastHPC propose au programmeur de faire une copie, sur les processus distants, des agents susceptibles d'y être utilisés. La synchronisation entre les processus s'effectue notamment dans trois cas [5] :

- lorsque les processus importent les copies des agents depuis d'autres processus pour maintenir la simulation dans un état cohérent
- pour mettre à jour ces copies
- pour effectuer la migration des agents entre les processus

Avec les outils de la plateforme, les programmeurs ont à définir un ensemble de méthodes nécessaires à la synchronisation des agents. La plateforme permet d'implémenter le mode de dépendance de données *ghost*, car la mise à jour des agents copie s'effectue par une communication collective en fin de pas de temps. La synchronisation est donc conservative. Même si RepastHPC fournit des fonctionnalités avancées en terme de planification de tâches et d'exécution des agents, la plateforme ne supporte pas la synchronisation optimiste pour le moment.

Flame. — Dans la plateforme Flame, tous les échanges entre les agents se font par messages, la synchronisation conservative repose donc sur celle de tableaux de

messages [4]. Elle est effectuée en deux étapes : la demande de synchronisation puis l'exécution de la synchronisation. Dans un premier temps, lorsqu'un processus a terminé d'exécuter ses agents, il verrouille son tableau de messages et envoie aux autres processus une demande de synchronisation. Après cette étape, il est encore possible de faire des actions qui ne nécessitent pas l'utilisation du tableau de messages. Lorsque tous les processus ont verrouillé leurs tableaux de messages, une seconde étape d'exécution de la synchronisation est effectuée par échange de messages entre les tableaux. Après ces deux étapes, les tableaux de messages sont débloqués et la simulation se poursuit. Puisqu'il n'y a pas de modification des données entre deux synchronisations des tableaux de messages, la plateforme Flame repose sur un mode *ghost*.

Pandora. — Dans la plateforme Pandora la synchronisation est conservative [20] et repose sur une grille 2D. Les données et les agents situés dans les zones de recouvrement sont copiés et envoyés aux cellules voisines à chaque pas de temps. Pour résoudre le problème de la dépendance de données, la simulation est découpée en parties, numérotées de 0 à 3. Au cours d'un pas de temps tous les processus exécutent séquentiellement chacune des parties, dans le même ordre : la partie 0, puis 1, 2 et 3. Une fois l'exécution d'une partie terminée, les zones de recouvrement sont envoyées aux cellules voisines. De cette façon, il n'y a pas de conflits de cohérence car les parties exécutées en parallèle ne sont pas adjacentes. Cette approche originale réduit les coûts de synchronisation avec un modèle en partie *ghost* et en partie *non ghost*, suivant l'emplacement des données. Cela limite tout de même l'utilisation de Pandora à des modèles spatiaux à deux dimensions.

Synthèse. — Les PDMAS existants utilisent principalement une approche conservative quant à la causalité, certains laissant cette gestion au développeur. Ceci paraît justifié car l'approche optimiste convient peu aux systèmes multi-agents qui animent les agents par pas de temps, donc de manière uniforme au sein de processus parallèles à l'inverse des systèmes à événements discrets où les processus ont une répartition temporelle des événements différente entre eux. Il est donc moins intéressant de faire avancer plus vite certains processus et une politique de répartition de la charge peut s'avérer plus efficace.

Les plateformes proposent principalement un mode *ghost*, moins lourd qu'une mise à jour systématique, qui recopie les données vers les autres processus au changement de pas de temps, lorsque tous les agents sont dans un état fixe.

Une analyse de plusieurs types de modèles permet cependant de montrer que tous n'ont pas les mêmes besoins de synchronisation. Nous proposons donc, dans la section suivante, une étude des besoins de synchronisation de modèles multi-agents et différentes politiques pouvant répondre à ces besoins.

3. Impact des synchronisations

La synchronisation est un point clé pour une exécution efficace d'une simulation multi-agents distribuée, du fait des nombreux échanges et dépendances temporelles

qu'elle induit. Son importance dépend néanmoins du modèle lui-même : dans un modèle où les agents n'interagissent pas, aucune synchronisation n'est nécessaire. Mais l'intérêt des modèles agents repose justement sur la capacité des agents à interagir [9]. Notre objectif est donc d'étudier l'impact de la synchronisation sur les temps d'exécution et l'impact des politiques de synchronisation plus relâchées. En effet, dans les systèmes multi-agents, l'observation d'un phénomène s'effectue en général au niveau macroscopique et non microscopique. De ce point de vue, il est possible que, dans les simulations composées d'un grand nombre d'agents, des synchronisations erronées ou fausses se compensent et limitent ainsi l'impact d'une synchronisation relâchée. Il est donc intéressant de mettre en relation l'erreur possible avec le surcoût dû à la synchronisation. Nous proposons dans la suite différentes politiques de synchronisation et étudions leur impact.

3.1. Quand synchroniser ?

Pour garantir l'accès à des données à jour et la cohérence des actions, en respectant les règles du modèle, plusieurs étapes de synchronisation sont nécessaires au cours d'une exécution distribuée, que la simulation soit en mode *ghost* ou *non ghost* :

- à la fin de chaque pas de temps pour permettre le passage au pas de temps suivant et garantir que tous les processus exécutent le même pas de temps.
- à la migration d'un agent d'un processus à un autre pour continuer à exécuter ses comportements.
- à la mise à jour des zones de recouvrement pour garder la continuité des champs de perception des agents lorsque l'environnement est distribué sur plusieurs processus.

Dans les modèles agents, il existe des modèles qui nécessitent que les agents accèdent aux données de l'environnement ou d'autres agents uniquement en lecture, d'autres en lecture et en écriture. La distribution de ces données sur plusieurs processus suppose d'en gérer les accès à distance et de synchroniser ces derniers pour garantir la cohérence de ces données.

Ainsi une exécution en mode *ghost* garantit uniquement la cohérence des modèles sans écriture concurrente, c'est à dire où deux entités ne modifient pas la même donnée au cours d'un pas de temps. En effet, utiliser le mode *ghost* avec des écritures concurrentes peut conduire à une violation des règles du modèle. Par exemple, dans un modèle proie-prédateur, plusieurs prédateurs pourraient manger une même proie au cours d'un pas de temps puisque, suite à une première attaque, la mort de la proie est enregistrée dans la copie de travail et non dans la copie *ghost* qui est utilisée pour connaître l'état du système. C'est seulement au changement de pas de temps que la mort de la proie sera reportée dans la copie *ghost*. À noter que ceci est vrai, même si la simulation n'est pas parallèle.

À l'opposé, dans le mode *non ghost*, les informations sont accessibles en lecture et en écriture. Ceci nécessite alors des mécanismes de synchronisation supplémentaires

pour les données détenues par d'autres processus ou situées dans les zones de recouvrement. Dans ce cas, les points de synchronisation précédemment définis ne sont pas suffisants car ils ne permettent pas de gérer les écritures dans les zones de recouvrement. L'accès aux données doit donc être géré pour garantir qu'aucune incohérence (ou biais) n'est injectée dans la simulation et que les informations sont à jour. Cela revient à introduire une politique de synchronisation des données à l'intérieur du pas de temps.

Les besoins en lecture / écriture des modèles agents étant variables, la synchronisation est à considérer au cas par cas. Pour illustrer ceci, nous analysons différents types de modèles dans la suite.

3.2. Analyse de modèles

Pour évaluer l'impact de la synchronisation sur les résultats d'exécution des simulations multi-agents, nous utilisons trois modèles agents (Proie-prédateur, Virus et Flocking) qui nécessitent des niveaux de synchronisation différents pour s'exécuter de manière cohérente.

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

Le modèle Flocking [27]. — Le modèle Flocking simule le vol d'une nuée d'oiseaux afin d'étudier le comportement collectif. Le modèle est composé d'un seul type d'agent, les oiseaux qui sont localisés dans l'espace et ont une zone de perception réduite. Chaque agent oiseau a trois comportements :

(1) la cohésion qui le pousse à se rapprocher des oiseaux proches
(2) l'alignement qui le pousse à se déplacer dans la même direction que les oiseaux voisins
(3) la séparation qui le pousse à tourner pour éviter un oiseau qui est trop près.

Ces comportements déterminent la nouvelle position de l'oiseau en fonction de la position des oiseaux qui composent son voisinage. Le modèle possède plusieurs paramètres : la taille de l'environnement, la distance maximale qu'un oiseau peut parcourir par pas de temps, la durée d'un pas de temps, les taux de cohésion, d'alignement et de séparation.

Ce modèle fonctionne en mode *ghost*. Les agents calculent leur déplacement en fonction de la position des oiseaux voisins obtenue au pas de temps précédent. La mise à jour des données des zones de recouvrement à chaque pas de temps garantit que chaque agent oiseau dispose des informations correctes pour calculer son déplacement.

Il n'y a donc pas de problème de concurrence d'accès sur ces données puisqu'elles sont accédées uniquement en lecture.

Le modèle Virus [28]. — Le modèle Virus permet de simuler la transmission et la survie d'un virus dans une population [30]. Il est composé d'un seul type d'agent : les personnes qui sont localisées sur une grille en deux dimensions et n'ont qu'une connaissance partielle de l'environnement dans lequel ils évoluent. Les agents ont cinq comportements :

(1) le vieillissement, jusqu'à ce qu'ils meurent
(2) le déplacement de manière aléatoire sur l'environnement
(3) l'infection des personnes de leur voisinage
(4) la récupération qui permet à un agent infecté de devenir immunisé avec une certaine probabilité
(5) la reproduction, pour les personnes non contaminées, qui renouvelle la population.

Le modèle possède plusieurs paramètres : la capacité de transport du virus, l'âge maximum, le taux de natalité, le taux de reproduction et le nombre de personnes porteuses du virus à l'initialisation du modèle.

Le modèle Virus fonctionne en mode *non ghost*. Il est donc possible que, dans un même pas de temps, un agent A, qui a été infecté, infecte à son tour un agent B. Si les deux agents ne sont pas exécutés sur le même processus, il est nécessaire de mettre à jour les données distantes. Pour finir, nous pouvons remarquer qu'un agent ne change pas son état même s'il est infecté plusieurs fois. Cette propriété, que nous appelons écriture idempotente, fait que nous n'avons pas à gérer de concurrence en écriture sur le changement de l'état de l'agent, puisque même si deux agents infectent un même agent l'ordre des deux exécutions n'a pas d'incidence sur le résultat final.

Le modèle Proie-prédateur [26]. — Le modèle Proie-prédateur explore la stabilité des écosystèmes. Le modèle étudié possède trois types d'agents : les loups (prédateurs), les moutons (proies/prédateurs) et l'herbe (proie). Les loups et les moutons se déplacent au hasard dans l'environnement. L'herbe disparaît lorsqu'elle est mangée et repousse après un temps fixé. Chaque étape coûte de l'énergie aux loups et aux moutons qui, lorsqu'ils n'en ont plus, meurent. L'énergie peut-être reconstituée pour un loup en mangeant un mouton et pour un mouton en mangeant de l'herbe. Pour permettre à la population de perdurer, les loups et les moutons ont une probabilité de se reproduire à chaque pas de temps. Tous les agents sont localisés sur une grille à deux dimensions et ne connaissent que leur zone de perception. Àchaque pas de temps, les agents loups et moutons exécutent les quatre comportements suivants dans l'ordre donné :

(1) se déplacer aléatoirement sur l'environnement
(2) se nourrir de proies si elles se situent dans leur champ de perception
(3) mourir s'ils n'ont plus d'énergie

(4) se reproduire.

Le modèle possède plusieurs paramètres : la taille de l'environnement, le nombre de loups, le nombre de moutons, le taux de natalité, le taux de reproduction, le gain de vie lorsqu'un prédateur mange une proie et le temps de croissance des agents herbe.

Le modèle Proie-prédateur fonctionne en mode *non ghost*. Dans la mesure où les prédateurs mangent les proies, ils en changent l'état, ce qui engendre une écriture dans les données de l'agent. Comme pour le modèle virus cela oblige une synchronisation pendant le pas de temps pour prendre en compte la modification mais il est, en plus, nécessaire de gérer les écritures concurrentes. Àcause du parallélisme, plusieurs prédateurs situés dans des processus différents peuvent être tentés de manger une même proie dans la zone de recouvrement. Si tous les prédateurs la mangent alors chacun bénéficiera d'un apport en énergie et donc d'une augmentation de sa durée de vie, ce qui constitue une erreur par rapport au modèle séquentiel. Il est donc nécessaire de synchroniser tous les agents qui souhaiteraient manger une proie pour garantir qu'un seul prédateur la mangera.

Synthèse. — L'analyse de ces modèles met en évidence différents besoins en termes de synchronisation. Ces besoins dépendent des caractéristiques du modèle et nous permettent de définir trois types en fonction des interactions entre les agents :

(1) les modèles en lecture (L), comme Flocking.
(2) les modèles en écriture idempotente (EI), comme le modèle virus, où l'état de l'agent ne change plus après une écriture.
(3) les modèles en écriture concurrente (EC), comme le modèle proie-prédateur.

Il est alors possible d'assister le modélisateur en lui fournissant le choix de la politique de synchronisation qui permet l'implémentation correcte de son modèle tout en limitant l'impact sur les performances. Ainsi nous définissons dans la section suivante plusieurs politiques de synchronisation.

3.3. Politiques de synchronisation

Nous proposons ici plusieurs politiques de synchronisation, issues de l'analyse des modèles précédents. Ces politiques sont définies et implémentées au sein de modes de synchronisation. Nous souhaitons évaluer l'impact de l'utilisation d'un mode de synchronisation sur les résultats et sur le temps d'exécution des simulations, sachant que d'autres modes pourraient présenter un intérêt pour d'autres modèles. Àces propositions nous ajoutons le cas « aucune synchronisation » qui sert de référence en ayant un coût de communication et synchronisation minimal.

Le mode **aucune synchronisation (NS)** distribue la simulation en n portions sans zone de recouvrement ni écritures distantes. Les agents peuvent se déplacer d'un processus à un autre en ayant un champ de perception tronqué lorsqu'ils sont proches des limites des processus.

Le mode **overlapping zones (OLZ)** ne gère que des zones de recouvrement, où des agents d'autres processus sont copiés et mis à jours à la fin de chaque pas de temps. Les écritures n'y sont pas reportées sur les originaux et sont écrasées au pas de temps suivant par la mise à jour. Ce mode est l'implémentation basique du mode *ghost*.

Le mode **écritures asynchrones (EA)** fait des écritures à distance sans attendre une confirmation ou une valeur de retour. Elle est utilisée lorsqu'un agent modifie une donnée de la zone de recouvrement et que cette écriture doit être prise en compte dans le pas de temps courant mais que l'agent n'attend pas de donnée en retour.

Le mode **synchronisation stricte (SS)** gère les zones de recouvrement et les écritures concurrentes distantes pour garantir au maximum la reproduction du cas séquentiel. Chaque demande en écriture est bloquante jusqu'à l'acquittement de sa prise en compte, ainsi la cohérence des données est garantie. Elle est ce qu'il y a de plus strict en termes de synchronisation sans revenir à une exécution séquentielle.

Contrairement à la synchronisation stricte, la **synchronisation stricte décalée (SSD)** s'effectue de manière non-bloquante. Ainsi, lorsqu'un agent effectue une demande de synchronisation, il est mis en attente de réponse jusqu'à la fin du pas de temps afin que l'exécution des autres agents se poursuive, assurant ainsi un meilleur recouvrement calcul-communication.

Pour diriger le choix d'un mode de synchronisation pour un modèle, il est nécessaire de connaître l'incidence qu'aura le choix de ce mode sur le modèle distribué. Les spécifications données précédemment permettent d'évaluer cette incidence et les propriétés garanties. Pour connaître l'impact de chacun des modes sur les performances de la simulation, nous présentons dans la suite les mesures de performance réalisées en implémentant ces modes sur les modèles étudiés. À noter que les modèles choisis sont tous spatialisés mais ces problématiques s'appliquent de la même manière sur des modèles non-spatialisés puisqu'elles sont liées aux échanges de données entre agents plutôt qu'à la position de ceux-ci. Par exemple, dans les modèles de représentation de réseaux sociaux, la distance entre deux agents peut être représentée par le nombre de sauts dans le graphe des connexions et ainsi servir de base pour grouper les agents proches et définir des zones de recouvrement sur lesquelles il faudrait appliquer les modes de synchrnisation.

4. Étude expérimentale de l'impact des modes de synchronisation

L'objectif de cette section est de mettre en évidence le lien entre les performances d'exécution d'un modèle et le mode de synchronisation choisi en fonction de la variation d'extensibilité ou de montée en charge. Comme aucune des plateformes vues précédemment n'offre de politique de synchronisation, nous les avons d'abord implémentées dans des modèles pour les tester.

4.1. Modèles et modes de synchronisation

Nous utilisons les trois modèles vus précédemment, choisis pour leurs différents besoins de synchronisation. La table 4.1 donne les modes de synchronisation utilisées avec chacun des modèles.

Modèle	Ghost	Type	Modes
Flocking	Oui	L	NS — OLZ
Virus	Non	EI	EA — OLZ — SSD
PP	Non	EC	OLZ — SSD — SS

Table 4.1. Modes de synchronisation utilisés

Le modèle Flocking est un modèle L. Deux modes de synchronisation sont donc testés : aucune synchronisation (NS) et utilisation de zones de recouvrement (OLZ), car le modèle de nécessite pas d'écriture. Le modèle Virus est un modèle EI. Les modes de synchronisation utilisés sont donc l'écriture asynchrone (EA), la synchronisation stricte décalée (SSD) et enfin les zones de recouvrement (OLZ) afin d'évaluer l'impact de l'absence d'écritures distantes sur les résultats du modèle. Le mode de synchronisation stricte (SS) n'est pas utilisé puisque l'écriture idempotente ne change pas l'état d'un agent contaminé. Il n'est donc pas nécessaire de gérer l'écriture concurrente sur son changement d'état puisque l'ordre d'exécution de deux écritures n'a pas d'incidence sur le résultat final. En revanche, le mode SSD qui implique un réordonnancement de l'exécution des agents est susceptible d'avoir un impact sur les résultats. Le modèle proie-prédateur est un modèle EC puisqu'une proie ne peut être mangée qu'une seule fois. Les modes de synchronisation utilisés sont la synchronisation stricte décalée (SSD), la synchronisation stricte (SS) et enfin les zones de recouvrement (OLZ) pour les mêmes raisons que pour le modèle Virus.

4.2. La plateforme de test

L'implémentation des modèles a été effectuée à partir d'une version expérimentale, notée 0.1, de notre plateforme FPMAS [19], une plateforme multi-agents parallèle qui repose sur la bibliothèque Zoltan pour gérer la distribution de la simulation. Dans FPMAS le modèle agent est représenté par un graphe afin de tirer parti des algorithmes de partitionnement parallèles de Zoltan. En réponse aux manques identifiés dans les plateformes existantes, comme cela a été souligné en 2.3, FPMAS a été conçue pour facilement introduire des mécanismes de synchronisation, et en particulier la synchronisation stricte. Puisque les plateformes existantes ne proposent pas de tels modes de synchronisation, il n'a pas été possible de mettre en place des courbes comparatives entre plateformes.

Une fois le modèle et ses modes de synchronisation implémentés, FPMAS peut l'exécuter dans un environnement parallèle adapté au calcul haute performance. La simulation est divisée en N portions, chacune associée à un processus. Les performances

dépendent alors du modèle exécuté. Ànoter que les modes de synchronisation sont implémentés directement dans les modèles, ce qui peut engendrer quelques différences de performances d'un modèle à l'autre. Pour cette raison nous ne faisons par la suite que des comparaisons entre modes par rapport à un modèle donné.

Pour exécuter les simulations, nous avons utilisé le Mésocentre de calcul de Franche-Comté. Le cluster est constitué de nœuds bi-processeurs, avec des processeurs Xeon E5 (8*2 cœurs) cadencés à 2.6 Ghz et 32 Go de mémoire vive. Le cluster possède un total de 1280 cœurs gérés par le système de batch SGE[(1)]. Les noeuds sont interconnectés par un réseau non bloquant QDR infiniBand[(2)] organisé en fat tree. Au cours des expériences conduites les variations de performances se sont avérées très faibles, entre 0.1 % et 0.6 %. Chaque point des courbes représente donc une moyenne de 10 exécutions avec 10 graines différentes.

4.3. Impact de l'extensibilité

L'extensibilité des modèles est étudiée en fixant le nombre d'agents de la simulation et en faisant varier le nombre de processus sur lesquels elle s'exécute. Nous mesurons l'impact de l'extensibilité à l'aide de deux métriques : le temps d'exécution et le speed-up.

Pour chacun des modèles nous avons calculé un speed-up avec comme référence le temps d'une exécution parallèle sur 16 cœurs car la taille des données des modèles est trop grande pour un seul processus. Le speed-up sur p processus est donné par $T(p_{ref})/T(p)$ où $T(p_{ref})$ est le temps d'exécution parallèle sur le nombre de processus de référence et $T(p)$ est le temps d'exécution sur p processus. Pour les trois modèles l'extensibilité est bonne avec un speed-up de plus ou moins 20 suivant les modèles, alors que le speed-up idéal est de 32. Les résultats obtenus par les modes sans synchronisation, ou avec moins de synchronisation, sont meilleurs que ceux ayant des synchronisations plus strictes.

Pour des raisons de reproductibilité, la même configuration initiale est utilisée pour toutes les exécutions, seule la graine varie d'une exécution à une autre.

Modèle Flocking. — L'environnement du modèle Flocking est basé sur un cube de $1\,000 \times 1\,000 \times 1\,000$ cellules. Àl'initialisation, 100 000 oiseaux sont répartis de manière aléatoire dans l'espace. Les taux de cohésion, de séparation et d'alignement sont fixés à 1 tandis que le taux d'aléa est lui fixé à 1.5. Ces paramètres ont été choisis afin de ne pas favoriser l'un des trois critères composant les comportements des oiseaux. Seul le taux d'aléa est supérieur aux autres taux dans le but de générer des déplacements plus chaotiques et donc de tester davantage la synchronisation. Chaque simulation est exécutée durant 2 000 pas de temps.

(1) `https://en.wikipedia.org/wiki/Oracle_Grid_Engine`
(2) `https://fr.wikipedia.org/wiki/Bus_InfiniBand`

La figure 4.1 donne les temps d'exécution du modèle Flocking pour les modes de synchronisation *NS* et *OLZ*. La différence des temps d'exécution entre les deux modes de synchronisation est d'environ 15 % pour 16 cœurs, 26 % pour 128 cœurs, et 65 % pour 512 cœurs. Lorsque le nombre de cœurs augmente, plus de messages sont nécessaires pour mettre à jour les zones de recouvrement, ce qui explique cette différence croissante.

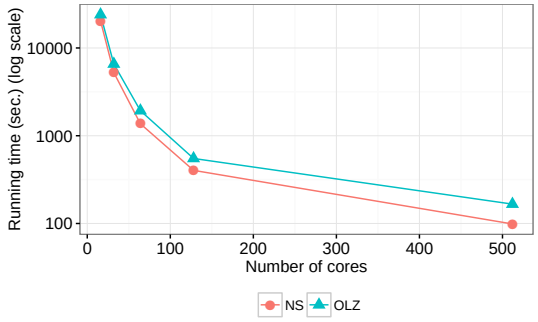


FIGURE 4.1. Temps d'exécution du modèle Flocking

MODÈLE VIRUS. — Le modèle Virus a été exécuté sur une grille de $1\,000 \times 1\,000$ cellules qui représente l'environnement avec une capacité maximale de 500 000 personnes. À l'initialisation 9 600 personnes sont saines et 640 sont infectées par le virus. Tous les agents sont positionnés de manière aléatoire sur l'environnement. Le taux d'infection est fixé à 0.65 et le taux de reproduction est fixé à 0.2. Le taux de récupération, c'est à dire le fait qu'une personne infectée devienne immunisée, est fixé à 0.5. Ces valeurs sont issues du modèle Virus de NetLogo. Seule la taille de l'environnement et la capacité maximale ont été adaptées pour obtenir un modèle de grande taille. Pour finir, chaque simulation est exécutée durant 800 pas de temps.

La figure 4.2 donne les temps d'exécution du modèle virus pour les synchronisations *OLZ*, *EA* et *SSD*. Pour les courbes *OLZ* il n'y a pas de point pour 512 cœurs car nous n'avons pas eu assez de temps sur le calculateur pour réaliser l'expérimentation. Les résultats obtenus avec le modèle Flocking se confirment ici avec une différence de 27 % pour 16 cœurs, seulement 15 % pour 128 et 47 % pour 512 cœurs. Deux raisons expliquent ce surcoût : le traitement additionnel en fin de pas de temps de la synchronisation SSD et l'augmentation du nombre de messages nécessaires à la synchronisation due au plus grand nombre de cœurs. Cette figure nous montre également le coût induit par la synchronisation des agents. Pour 16 cœurs, les temps d'exécution de la courbe *OLZ* sont environ 8 fois meilleurs que la courbe *SSD*. Cette différence tend à décroître avec l'augmentation du nombre de cœurs, par exemple, pour 256 cœurs cette différence n'est plus que de 6.6. Pour les courbes *EA* et *SSD* le ratio n'est que de 1.27 pour 16 cœurs, et de 1.4 pour 512 cœurs. Il croît donc avec le nombre de cœurs du fait du nombre plus important de messages.

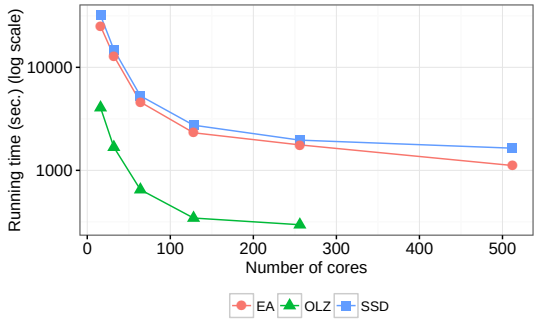


FIGURE 4.2. Temps d'exécution du modèle Virus

MODÈLE PROIE-PRÉDATEUR. — Le modèle Proie-prédateur (PP) utilisé pour les expérimentations est basé sur un environnement grille de 400×400 où 25 000 moutons et 17 000 loups sont initialement positionnés de manière aléatoire. Le modèle des comportements et l'initialisation de l'énergie des agents loups et moutons est issue du modèle NetLogo. L'énergie gagnée par un mouton lorsqu'il mange de l'herbe est fixée à 5 et à 20 lorsqu'un loup mange un mouton. En ce qui concerne les taux de reproduction, ils sont fixés à 0.5 pour les moutons et à 0.4 pour les loups. Le temps de croissance de l'herbe est de 8. Les simulations sont exécutées durant 2 000 pas de temps.

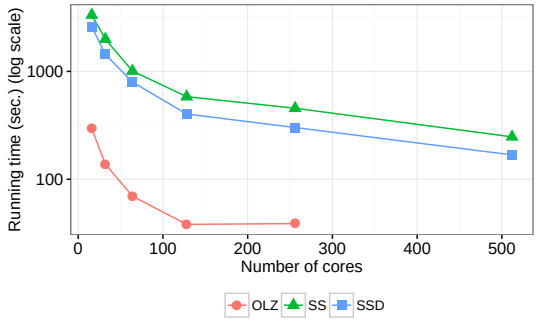


FIGURE 4.3. Temps d'exécution du modèle PP

La figure 4.3 présente les temps d'exécution du modèle PP pour les modes de synchronisation *OLZ*, *SSD* et *SS*. Pour la même raison que précédemment, aucun calcul n'a été effectué sur 512 cœurs avec *OLZ*. Les points de 16 à 64 cœurs permettent cependant de mettre en évidence, comme pour le modèle Virus, le coût important à payer pour avoir des modes de synchronisations strictes. La courbe *SSD* montre que la relaxation de la synchronisation de ce mode à la fin du pas de temps permet un gain de temps d'exécution par rapport à la synchronisation stricte. Ce gain est dû au fait que le traitement des agents n'est plus bloqué en attente de la réponse à une écriture concurrente : on a donc un meilleur recouvrement calcul-communication.

Sur les modèles étudiés, le calcul des speed-up montre une bonne extensibilité et le niveau de synchronisation choisi pour implémenter le modèle ne semble pas induire d'impact. Les simulations avec un grand nombre de cœurs profitent donc bien du parallélisme, ce qui confirme que les systèmes multi-agents peuvent bénéficier d'une parallélisation. Néanmoins, les modes de synchronisation ont un coût très important (jusqu'à un facteur 8), qui est dû aux communications engendrées. En effet, dans les modèles étudiés, les agents ont un comportement relativement simple qui s'exécute beaucoup plus rapidement qu'une communication même si le réseau utilisé pour les expériences est à très faible latence et haut débit.

4.4. Impact de la montée en charge

La montée en charge est réalisée en fixant le nombre de processus et en faisant varier le nombre d'agents. Le modèle Proie-prédateur n'est pas présenté dans cette section car il est très difficile d'y faire varier le nombre d'agents puisque la population s'auto-équilibre.

Modèle Flocking. — Le jeu de valeurs utilisé pour évaluer la montée en charge du modèle Flocking est le même que précédemment, à la différence que le nombre d'agents oiseaux qui composent la simulation varie de 100 000 à 1 000 000.

La figure 4.4 présente l'impact de la montée en charge sur 512 cœurs. Les modes *NS* et *OLZ* supportent bien la charge jusqu'à 500 000 agents. Au delà de 500 000 agents, les courbes croissent plus rapidement. Sans surprise, la version sans synchronisation supporte mieux la charge que la version avec zone de recouvrement, qui consomme environ un tiers de performance en plus. L'accroissement de la charge à partir de 500 000 agents s'explique par un manque d'optimisation lors de la recherche du voisinage des agents. Au lieu de mettre à jour uniquement les agents qui arrivent ou partent du voisinage, un parcours de l'environnement est effectué à chaque pas de temps pour établir le champ de perception de chaque agent.

Figure 4.4. Montée en charge du modèle Flocking

Modèle Virus. — Les paramètres utilisés pour la montée en charge sont les mêmes que précédemment.

La figure 4.5 montre l'impact de la montée en charge du modèle Virus de 100 000 à 700 000 agents et 800 pas de temps. Seulement 64 cœurs ont été utilisés pour cette courbe car il n'a pas été possible d'obtenir à nouveau autant de ressources de calcul (512 cœurs) sur le cluster partagé.

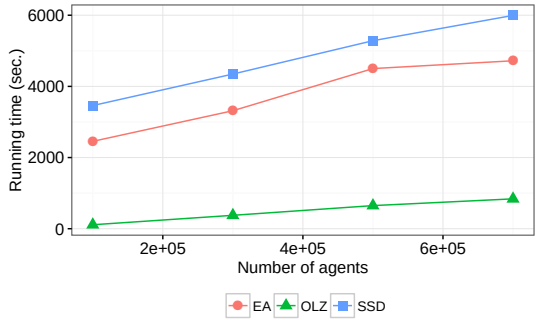


Figure 4.5. Montée en charge du modèle Virus

La courbe *OLZ* supporte mieux la montée en charge puisqu'elle ne gère pas les écritures. Pour les autres, on constate que *EA* supporte mieux la charge que *SSD*. La courbe *SSD* reste linéaire, alors que la courbe *EA* croît très peu de 500 000 à 700 000 agents. Ceci est dû au surcoût lié à l'accumulation des synchronisations en fin de pas de temps. Le ratio de performance obtenu entre 100 k et 700 k pour la courbe *EA* est de 1.9, 1.73 pour *SSD* et 7.5 pour *OLZ*.

Les résultats sur la montée en charge confirment que, quelle que soit le mode de synchronisation utilisé, l'exécution distribuée est bénéfique pour les modèles multi-agents. Les résultats obtenus avec le mode de synchronisation *OLZ* montrent que ce mode est plus efficace que les modes plus contraints. Nous avons donc étudié l'impact des modes de synchronisation sur les résultats des exécutions, ce que nous présentons dans la suite.

4.5. Impact sur les résultats

Pour étudier l'impact des différents modes, nous observons deux résultats : le résultat de la simulation lui-même et le nombre d'interactions incohérentes. Le résultat d'une simulation est ce qui est attendu par le modélisateur. Nous étudions l'impact d'une exécution plus ou moins synchronisée sur les résultats de l'exécution. Par ailleurs, dans les cas où les écritures dans les zones de recouvrement sont gérées, c'est à dire avec les modes *EA*, *SS* et *SSD*, des interactions peuvent mettre en évidence une incohérence entre l'information locale et l'information distante. Par exemple pour le modèle Proie-prédateur, une incohérence peut apparaître si un mouton est vivant dans la représentation locale à un processsus de la zone recouvrement alors qu'il est mort

dans le processus distant qui possède le mouton. En conséquence, prendre seulement en compte l'état local de la zone de recouvrement mène à une action incohérente si un loup du premier processus mange ce mouton. De même, pour le modèle Virus, un agent essaie d'en infecter un autre seulement si ce dernier est sain. Ainsi, pour analyser l'impact des interactions sur les résultats des simulations nous comptabilisons le nombre total de demandes de synchronisation, les synchronisations pour lesquelles l'information était cohérente (notées *CO*) et celles pour lesquelles l'information était incohérente (notées *NCO*). Cette analyse n'inclut pas le modèle Flocking car elle ne s'applique pas à un modèle en lecture.

À noter que nous souhaitons ici mettre en évidence l'impact des modes de synchronisation sur les résultats des exécutions. Or, que ce soit dans le cas du modèle virus ou du modèle proie-prédateur, faire la moyenne des résultats de plusieurs exécutions pourrait conduire à un lissage des erreurs, une erreur pouvant en compenser une autre. Nous avons donc fait le choix ici, pour la mise en évidence de la problématique, de comparer des exécutions uniques (avec la même graine) pour chacun des modes de synchronisation. Ces expériences permettent donc de mettre en évidence que la problématique existe : les résultats diffèrent entre deux exécutions avec des modes de synchronisation différents. Il faudrait plus d'expériences, et cette fois un calcul de moyenne ou d'indicateurs statistiques pour la quantifier.

Modèle Virus. — La figure 4.6 présente le détail des demandes de synchronisation pour les synchronisations *EA* et *SSD*.

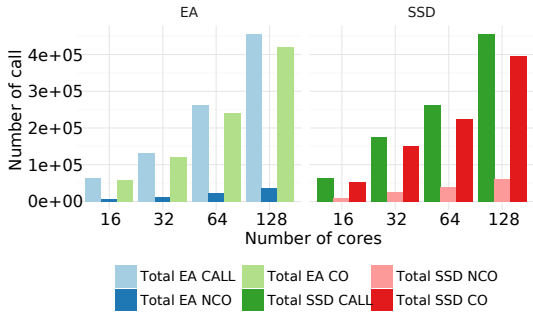
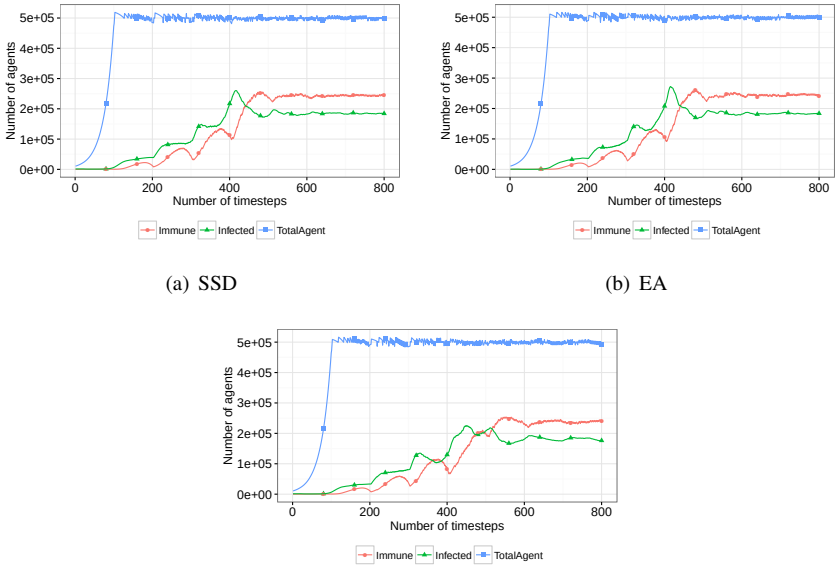


Figure 4.6. Détail des synchronisations du modèle Virus

Pour les deux modes de synchronisation le nombre total de demandes de synchronisation croît de manière linéaire avec le nombre de cœurs car, lorsque le nombre de cœurs augmente la simulation est divisée en plus de processus. Il y a donc plus de zones de recouvrement et plus d'interactions potentielles. Le nombre d'interactions *NCO* est très faible pour *EA*. Il croît d'environ 10 % pour *SSD* à cause des synchronisations gérées en fin de pas de temps. Les agents qui effectuent une demande de synchronisation sont suspendus et exécutés à la réception de la réponse, en fin de pas de temps. De ce

fait, certaines interactions qui étaient potentiellement *CO* au moment de la demande de synchronisation peuvent devenir *NCO* si les agents concernés n'avaient pas encore été exécutés. Il faut donc poser la question de savoir quel est l'impact de ces interactions *NCO* sur les résultats.

La figure 4.7 présente les résultats de l'exécution sur 128 cœurs. Les résultats pour *SSD* (figure 4.7(a)) et *EA* (4.7(b)) sont quasi identiques. Les quelques variations s'expliquent par les interactions *NCO* traitées en fin de pas temps par le mode *SSD*. En revanche, les résultats de *OLZ* (figure 4.7(c)) présentent de nombreuses différences, malgré une tendance identique.



(a) SSD

(b) EA

(c) OLZ

FIGURE 4.7. Résultats d'une exécution du modèle Virus

MODÈLE PROIE-PRÉDATEUR. — La figure 4.8 présente le détail des synchronisations lors de l'exécution du modèle PP. Comme pour le modèle Virus, le nombre de demandes de synchronisation croît de manière linéaire avec le nombre de cœurs. Le nombre d'appels *NCO* reste plus faible pour le mode *SS* que pour le mode *SSD*.

La figure 4.9 présente les résultats sur 128 cœurs des modes *SSD*, *SS* et *OLZ*. Le modèle PP est un modèle très sensible : la modification d'un paramètre peut conduire à une instabilité qui se traduit par la mort d'une des espèces. Le changement de mode induit ainsi des différences de résultats entre les courbes *SSD* (figure 4.9(a)) et les courbes *SS* (figure 4.9(c)). La figure 4.9(b) qui représente l'exécution sans synchronisation montre des résultats proches du mode *SSD* (figure 4.9(a)). Nous retrouvons tout de même des courbes cycliques qui tendent à s'équilibrer.

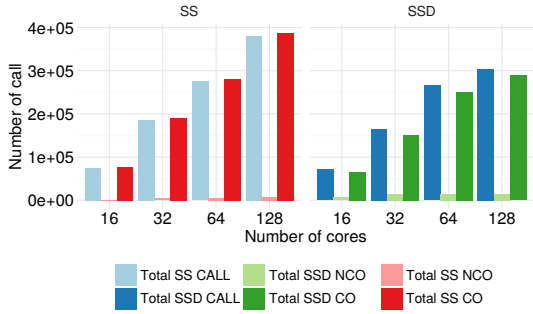
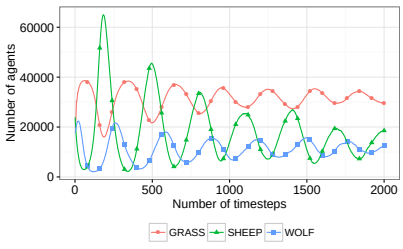
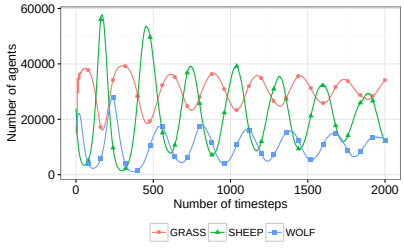


FIGURE 4.8. Détail des synchronisations du modèle PP

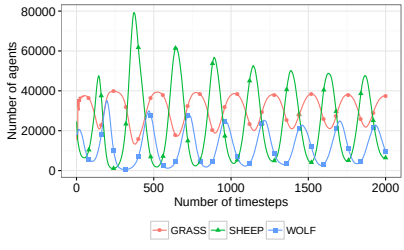
Au vu des courbes, on peut s'interroger sur ce qu'il est important d'observer dans les résultats du modèle. Si nous regardons plus en détails ces figures, nous remarquons que le nombre maximal de moutons diffère, environ 60 000 pour *OLZ* et *SSD* et environ 80 000 pour *SS*. On remarque également que sur la figure 4.9(c) (synchronisation *SS*) la phase est régulière avec un décalage de l'ordre de $\pi/2$. En revanche, sur les figures 4.9(b) et 4.9(a), un décalage de phase s'effectue au cours du temps. Les courbes *OLZ* et *SSD* sont semblables et donnent des résultats erronés.



(a) SSD



(b) OLZ



(c) SS

FIGURE 4.9. Résultats d'une exécution du modèle PP

Les résultats présentés ici ont été obtenus grâce à la version expérimentale de FPMAS (version 0.1) destinée à évaluer l'utilisation d'une structure à base de graphes sur la distribution des modèles. Cette version expérimentale n'offrant pas de support natif pour la synchronisation, les modes de synchronisation ont du être implémentés directement dans le code de chaque modèle pour être évalués. Or les résultats mettent en évidence la nécessité de traiter avec précaution les problématiques de synchronisation, inhérentes aux interactions nécessaires à l'exécution du modèle. Nous avons donc défini une interface générique des modes de synchronisation, rendant possible l'implémentation de divers modes dans la plateforme FPMAS ainsi que leur application à n'importe quel modèle. Cette interface est présentée dans la section suivante.

5. Interface de synchronisation générique

Dans la version 1.1 de la plateforme FPMAS, l'idée est d'offrir au modélisateur, de manière native, les outils nécessaires à une distribution implicite des interactions pour faciliter l'implémentation en donnant un résultat exact.

Dans cette section nous présentons donc une définition formelle et générique d'un mode de synchronisation dans le contexte de la simulation distribuée de systèmes multi-agents, ainsi que des implémentations possibles afin, notamment, de généraliser les modes de synchronisation décrit dans la Section 3 à n'importe quel SMA. Certains modes sont actuellement implémentés dans la version 1.1 de FPMAS, disponible sur GitHub[(3)].

5.1. Représentation à base de graphes des SMA

Comme pour la version expérimentale, FPMAS 1.1 est basé sur le concept de *graphes distribués* [1] pour représenter et simuler des SMA. Un des intérêts principaux d'une telle représentation est la possibilité de distribuer automatiquement et de manière optimisée les simulations grâce à des algorithmes de partitionnement de graphe, ce qui a déjà été validé dans nos travaux précédents [1, 19]. Mais la structure de graphe présente également l'avantage d'expliciter les interactions, comme dépendances de données, entre les agents : on suppose que deux agents dans la simulation vont être amenés à interagir, c'est à dire à échanger des données, si et seulement si il existe au moins un lien entre les nœuds du graphe représentant ces agents. Une telle restriction permet de faciliter la gestion des problèmes de synchronisation, car les communications auront naturellement lieu de manière distribuée, en se limitant au voisinage local de chaque nœud.

La structure de graphe distribué est cependant une structure de données relativement bas niveau facilitant la distribution et la synchronisation de la simulation, qui ne vise pas à limiter les modèles simulés. Par exemple, dans FPMAS, les modèles à base de grille sont eux mêmes définis à partir d'un graphe. Dans ce contexte, une grille peut être définie comme un ensemble de cellules connectées entre elles afin d'exprimer les

[(3)] `https://github.com/FPMAS/FPMAS`

dépendances spatiales. La position des agents, eux-mêmes représentés par des nœuds, peut être représentée par un lien entre l'agent et une cellule. Un algorithme distribué permet de construire dynamiquement les liens entre un agent et ceux localisés dans son champ de perception, afin de lui permettre d'interagir avec eux.

FPMAS fournit également des fonctionnalités permettant de construire automatiquement le graphe représentant une grille à partir de sa forme et de la position des agents spécifiées par l'utilisateur. La structure sous-jacente de graphe est ainsi rendue invisible, tout en permettant l'utilisation de nos modes de synchronisation sur un modèle à base de grille.

Les modes de synchronisation peuvent donc être définis au niveau du graphe le plus général : par conception, ils seront alors applicables sur tout modèle, qu'il soit basé sur une grille ou un graphe arbitraire.

À noter que RepastHPC permet également de définir des dépendances entre agents grâce à un graphe, afin de mettre à jour automatiquement les *copies* des dépendances exécutées sur d'autres processus. En revanche, seule la synchronisation en mode *ghost* est possible, et RepastHPC ne fournit pas directement de fonctionnalité de distribution de la simulation à partir de ce graphe, même s'il est possible de s'interfacer relativement facilement avec les librairies de partitionnement de graphe existantes [6].

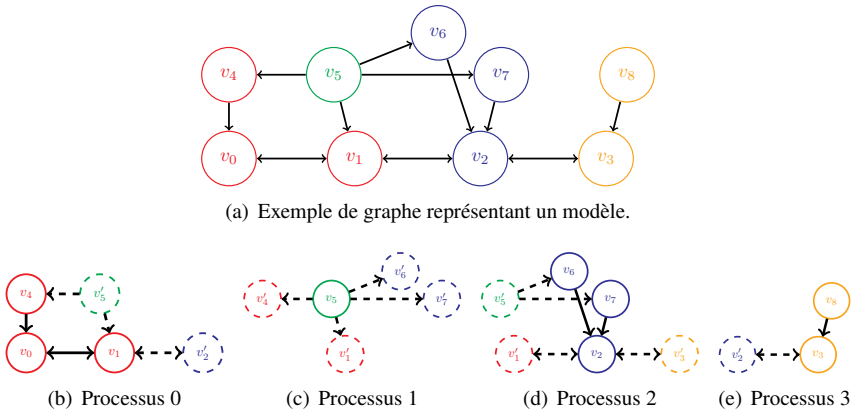
5.2. Distribution du graphe

Nous considérons dans cette section un modèle arbitraire déjà représenté par un graphe : les agents en sont les nœuds et les arcs les interactions possibles. Le partitionnement de ce graphe associe un ensemble de nœuds à chaque processus, appelés les nœuds *locaux* par rapport au processus, qui est alors chargé d'exécuter les agents représentés par ces nœuds. Afin d'assurer la continuité du modèle malgré la distribution, les interactions des agents *locaux* avec des agents *distants* sont permises par l'ajout de nœuds *distants* dans le graphe local et d'arcs reliant les nœuds *locaux* aux nœuds *distants*.

Un exemple de distribution du graphe sur 4 processus est présenté figure 5.1. Une couleur est associé à chaque processus. Les nœuds pleins sont *locaux*, les pointillés sont *distants*.

L'intérêt d'une telle structure est que le voisinage de chaque nœud *local* est préservé, quel que soit la distribution du modèle. On peut ainsi se permettre de définir une interface **générique** pour l'accès aux données de chaque noeud, qu'il soit *local* ou *distant*, afin d'abstraire cet aspect de la distribution à l'utilisateur.

Le concept de nœuds *distants* est très similaire aux *copies* des agents réalisées par RepastHPC. D'autre part, pour un modèle à base de grille, on peut légitimement considérer les cellules *distantes* dans le graphe comme une généralisation du concept de « zones de recouvrement » utilisées par d'autres plateformes de simulation distribuée.



(a) Exemple de graphe représentant un modèle.

(b) Processus 0 (c) Processus 1 (d) Processus 2 (e) Processus 3

FIGURE 5.1. Exemple de distribution du graphe.

5.3. DÉFINITION DE L'INTERFACE DE SYNCHRONISATION

Une fois la simulation distribuée, chaque processus est donc responsable d'un ensemble de nœuds *locaux* et *distants*. Seuls les agents associés à des nœuds locaux sont exécutés, mais chacun d'eux peut interagir avec les nœuds *locaux* **et** *distants* auxquels ils sont connectés. FPMAS définit une interface générique pour gérer l'accès à n'importe quel nœud, qu'il soit *local* ou *distant*, de sorte à abstraire complètement la distribution du point de vue de l'implémentation du modèle. La gestion de l'accès aux données est alors assurée implicitement par le mode de synchronisation choisi pour exécuter le modèle.

On définit ainsi l'interface `SynchronizationMode`, elle-même constituée de 3 interfaces :

- `Mutex` : gère l'accès aux données des nœuds.
- `DataSync` : gère la synchronisation des données des nœuds en fin de pas de temps.
- `SyncLinker` : permet de dynamiquement créer ou supprimer des nœuds et des liens dans le graphe.

Le diagramme de classes associé au graphe distribué et aux modes de synchronisation est présenté figure 5.2.

L'interface `Mutex`, pour *Mutual Exclusion*, définit notamment les méthodes `read()` et `acquire()` qui permettent d'accéder aux données de chaque nœud respectivement en lecture ou écriture de manière concurrente, indépendamment du processus sur lequel le nœud est effectivement localisé. Les méthodes `releaseRead()` et `releaseAcquire()` doivent être appelées lorsque le travail sur le nœud est terminé.

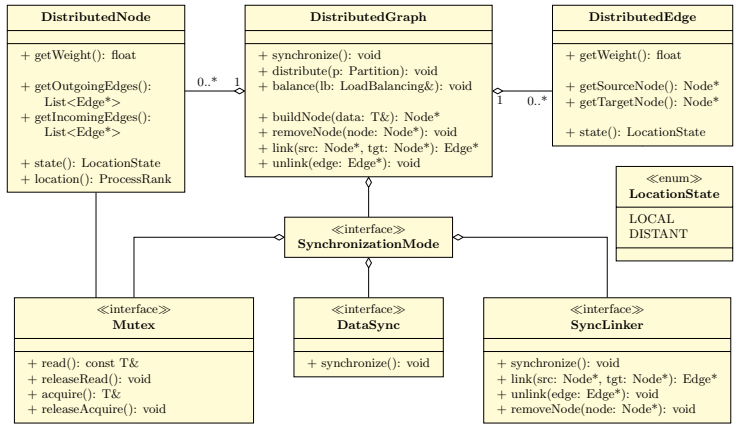


FIGURE 5.2. Diagramme de classe associé au graphe distribué

L'interface `DataSync` ne définit qu'une seule méthode : `synchronize()`. Le comportement de cette méthode est complètement déterminé par le mode de synchronisation implémenté. La seule garantie associée est l'appel automatique de cette méthode par FPMAS à la fin de chaque pas de temps par tous les processus.

Il est important de noter qu'aucune contrainte n'est imposée à propos de la synchronisation des données des noeuds au niveau de l'interface `SynchronizationMode`. Par exemple, il n'est pas garanti à ce niveau que l'appel des méthodes `Mutex::acquire()` et `DataSync::synchronize()` permette effectivement d'effectuer des écritures sur des données *distantes*. Comme énoncé dans la Section 3, ce n'est effectivement pas le cas pour la synchronisation **Overlapping Zones**, contrairement à la **Synchronisation Stricte**. Pourtant, les appels aux méthodes génériques de l'interface `SynchronizationMode` restent les mêmes, quel que soit le mode de synchronisation utilisé.

L'interface `SyncLinker` permet la gestion de la structure de graphe, ce qui implique la création de liens (`link()`), la suppression de liens (`unlink()`) ou la suppression de nœuds (`removeNode()`). Comme ces opérations peuvent s'appliquer à des noeuds *distants*, une synchronisation des processus est nécessaire pour les prendre en compte. Une méthode `synchronize()` est ainsi définie, et son appel automatique est également garanti à la fin de chaque pas de temps par tous les processus. Le comportement de chaque méthode dépend du mode de synchronisation implémenté. En revanche, il doit être garanti que chaque opération soit prise en compte au plus tard au moment du retour de la méthode `synchronize()`. L'implémentation de l'interface `SynchronizationMode` doit donc prendre en charge la création/suppression dynamique de liens et de nœuds, y compris dans le cas ou au plus un nœud *distant* est impliqué dans l'opération. De telles fonctionnalités sont effectivement nécessaires pour maintenir un état cohérent du graphe distribué au cours de la simulation.

L'implémentation d'un modèle avec FPMAS se base ainsi sur des appels à ces méthodes abstraites, indépendamment de la distribution du modèle ou du mode de synchronisation actuel. Cependant, le comportement réel de chaque méthode est déterminé au moment de l'exécution, en fonction du mode de synchronisation choisi. Comme démontré dans la Section 4, les résultats du modèle peuvent alors varier. La section suivante présente des exemples d'implémentation des différents composants de l'interface `SynchronizationMode`.

5.4. Spécification des modes de synchronisation

L'objectif de cette section est de présenter différentes implémentations possibles de l'interface `SynchronizationMode` à partir des modes de synchronisation définis à la Section 3. Cette liste n'est pas exhaustive, et plusieurs implémentations peuvent exister dans la même librairie : il suffit de changer un paramètre dans la définition du modèle pour sélectionner le mode de synchronisation utilisé pour l'exécuter.

Pour le moment, seuls les modes **Synchronisation Stricte** et **Overlapping Zones** ont été implémentés dans FPMAS, avec les classes respectives `HardSyncMode` et `GhostMode`. En revanche, il est déjà possible d'élaborer une spécification permettant d'implémenter les interfaces précédentes pour les autres modes.

GhostMode. — Dans ce mode de synchronisation, les données *distantes* sont seulement mises à jour à la fin de chaque pas de temps. Les écritures effectuées sur les données *distantes* ne sont pas reportées au processus d'origine, et sont écrasées en fin de pas de temps.

- `Mutex` : les méthodes `read()` et `acquire()` renvoient simplement une référence vers la donnée locale. L'état des données locales correspond aux données importées à la fin du pas de temps précédent, avec les éventuelles modifications effectuées localement au cours du pas de temps. Pour les données *locales*, les modifications sont bien prises en compte. En revanche, pour les données *distantes*, les modifications sont écrasées à la fin du pas de temps. Les méthodes `releaseRead()` et `releaseAcquire()` n'ont pas d'effet, car il n'est pas nécessaire de gérer la concurrence d'accès dans ce mode : en effet, les écritures distantes ne sont pas permises, et les agents sont actuellement exécutés de manière séquentielle sur chaque processus.
- `DataSync` : la méthode `synchronize()` met à jour tous les nœuds *distants* en important les données correspondantes sur les processus d'origine.
- `SyncLinker` : les opérations effectuées sur le graphe sont stockées pendant le pas de temps, et sont exportées lors de l'appel à la méthode `synchronize()`. Ce mode nécessite donc d'attendre la prochaine synchronisation avant de pouvoir utiliser les liens créés au cours du pas de temps, ou que les suppressions de liens ou de nœuds soient effectives, ce qui correspond bien aux garanties minimales imposées au niveau de l'interface `SynchronizationMode`.

HardSyncMode. — Les données des nœuds *distants* sont systématiquement importées depuis le processus d'origine de manière **bloquante**, en effectuant des *requêtes*, avec une gestion de la concurrence en lecture et en écriture, ce qui permet de reporter les écritures sur les données *distantes* au processus d'origine. Un algorithme de terminaison [8] place les processus en attente en fin de pas de temps jusqu'à ce que toutes les requêtes de tous les processus aient été traitées. En effet, dans un tel schéma d'exécution, même si un processus a a terminé l'exécution de ses agents, il se peut qu'un autre processus b ait besoin d'une donnée du processus a afin qu'il puisse lui même terminer l'exécution de ses agents. L'algorithme de terminaison permet aux processus ayant terminé l'exécution de leurs agents de répondre aux requêtes des autres jusqu'à ce que plus aucun processus n'ait de requêtes à effectuer et qu'ils aient tous terminé l'exécution de leurs agents.

- `Mutex` : les méthodes `read()` et `acquire()` envoient une requête au processus d'origine pour obtenir les données à jour. Les appels bloquent jusqu'à ce que les données soient disponibles : plusieurs processus peuvent lire simultanément la même donnée, mais un accès exclusif doit être assuré pour *acquérir* les données afin de pouvoir effectuer des écritures. La méthode `releaseRead()` indique simplement au processus d'origine que le processus local a terminé d'effectuer sa lecture, pour notifier que la donnée est à nouveau disponible pour d'autres processus. Le fonctionnement de `releaseAcquire()` est similaire, mais il est en plus nécessaire d'envoyer les modifications locales au processus d'origine. Le mécanisme de requêtes s'applique également dans le cas de l'accès à une donnée locale : en effet, le processus local doit attendre que ses propres données soit disponibles pour gérer la concurrence avec les autres processus.
- `DataSync` : la méthode `synchronize()` applique un algorithme de terminaison pour continuer à répondre aux requêtes des autres processus, jusqu'à ce que toutes les requêtes aient été traitées.
- `SyncLinker` : les modifications du graphes sont effectuées à la volée, de manière bloquante, au sein du pas de temps. La méthode `synchronize()` exécute également un algorithme de terminaison pour traiter toutes les requêtes avant de passer au pas de temps suivant.

Écriture Asynchrone. — Ce mode n'est pas implémenté dans la version 1.1 de FPMAS, la spécification fournie ici est donc théorique.

Dans ce mode de synchronisation, les écritures sont effectuées de manière **non bloquante**, sans gestion de la concurrence. Les données *distantes* accédées en lecture ou en écriture ne sont pas importées depuis les processus d'origine au cours du pas de temps. Les modifications appliquées localement à ces données sont tout de même renvoyées au processus d'origine au sein du pas de temps, afin de prendre en compte les écritures. En revanche, chaque écriture écrase systématiquement les précédentes au cours du pas de temps. Les données sont également mises à jour à la fin de chaque pas de temps, comme avec le `GhostMode`, afin d'assurer une actualisation partielle

des données. Ce mode permet de prendre en compte des écritures distantes, avec un impact moindre sur les performances comparé à `HardSyncMode`, car les écritures sont réalisées de manière non bloquante, sans gestion de la concurrence : son utilisation est donc plus limitée que `HardSyncMode`, car des incohérences peuvent être induites pour certains modèles.

- `Mutex` : Les méthodes `read()` et `acquire()` renvoient une référence vers la donnée locale, comme pour `GhostMode`. La méthode `releaseRead()` n'a pas d'effet particulier, mais `releaseAcquire()` envoie de manière non-bloquante les modifications locales au processus d'origine.
- `DataSync` : La méthode `synchronize()` applique un algorithme de terminaison pour s'assurer que toutes les écritures distantes aient été reçues. Les données *distantes* sont également mises à jour à la manière de `GhostMode`.
- `SyncLinker` : implémentation identique à celle du `GhostMode`.

Le modèle Virus est un bon exemple d'utilisation possible de ce mode. A chaque pas de temps, les agents vérifient l'état des agents à proximité, infectés ou sains. Si l'agent est déjà infecté, il est inutile de l'infecter à nouveau, sinon on l'infecte, ce qui représente ici une écriture. Dans le cas où l'agent voisin est *distant* :

- Si aucune autre écriture n'a été effectuée sur le processus local, on accède à l'état de l'agent au pas de temps précédent. S'il était déjà infecté, il est inutile de chercher à l'infecter à nouveau et aucune communication n'est effectuée. S'il ne l'était pas, l'agent est infecté en effectuant une écriture non bloquante, sans aller vérifier l'état réel de l'agent sur le processus d'origine : peut-être a-t-il déjà été infecté au sein du pas de temps depuis un autre processus, mais dans ce cas l'écriture actuelle ne modifie pas son état final. De plus, l'infection est prise en compte au plus tôt sur le processus d'origine, ce qui pourra éviter à d'autres agents sur ce processus d'effectuer inutilement des écritures.
- Si une écriture a déjà été effectuée sur le processus actuel, l'état de la donnée indique que l'agent a été infecté par un autre agent *local* et il est donc inutile d'effectuer l'écriture à nouveau.

Dans tous les cas, l'agent *distant* est bien infecté, mais les communications sont minimisées par rapport à `HardSyncMode`, qui aurait produit le même état final.

Synchronisation Stricte Décalée. — Cette synchronisation est similaire à `HardSyncMode`, sauf que cette fois les requêtes doivent être réalisées de manière **non bloquante** : on peut poursuivre l'exécution des autres agents pendant que l'un d'eux est en attente pour accéder à des données *distantes*. Les agents en attente sont alors exécutés à la fin du pas de temps. Ce mode ne peut être implémenté dans la version actuelle de FPMAS, car il nécessite de modifier le schéma d'exécution des agents, ce qui n'est pas possible depuis l'interface `SynchronizationMode`. En effet, les agents sont actuellement exécutés de manière séquentielle par le composant `Runtime`, qui détermine l'ordre (aléatoire) d'exécution des agents en début de pas de temps.

5.5. Synthèse

L'intérêt de l'approche proposée ici est de décorréler l'implémentation du modèle de l'implémentation distribuée des interactions et de leur sémantique, le code du modèle ne contenant que les fonctions génériques de synchronisation. Le concepteur de modèle peut alors se reposer sur FPMAS pour l'implémentation du mode de synchronisation adapté.

FPMAS permet, de plus, d'implémenter de nombreux modes de synchronisation, avec des comportements très différents notamment en termes de gestion des écritures, à partir de l'interface générique `SynchronizationMode`. On peut d'ailleurs imaginer des nuances aux modes précédents, par exemple en combinant la synchronisation des données de `HardSyncMode` avec la gestion du graphe de `GhostMode`. L'`Écriture Asynchrone` est également sujette à certaines variantes : on pourrait par exemple ne pas mettre à jour les données en fin de pas de temps, ou stocker les écritures pour les exporter seulement à l'appel de la méthode `synchronize()`.

Dans tous les cas, comme l'implémentation des modèles avec FPMAS se base sur des appels aux méthodes génériques de l'interface `SynchronizationMode`, il est trivial de tester chaque mode de synchronisation sur un modèle donné, afin de facilement évaluer l'impact sur les résultats ou les performances.

6. Conclusion

Notre étude sur l'impact des politiques de synchronisation vise à sensibiliser la communauté agent, et plus particulièrement les modélisateurs, aux problèmes liés aux dépendances de données lors de la parallélisation d'un modèle. Nous évaluons pour cela différents modes de synchronisation en lien avec différents modèles et nous mettons en évidence que, suivant le modèle à implémenter, le niveau de synchronisation a un impact sur les performances d'exécution et les résultats. Àtravers nos expérimentations, nous quantifions le coût de la synchronisation sur les performances de la simulation suivant les modèles et nous montrons son impact sur les résultats. La synchronisation est, de fait, dépendante des comportements des agents : c'est au modélisateur de prévoir l'exécution du modèle et donc d'adapter la modélisation pour prendre en compte le coût et l'impact de la synchronisation lors de la conception d'un modèle parallèle.

Nos travaux les plus récents ont permis de définir formellement une interface générique de synchronisation, qui permet d'implémenter divers modes de synchronisation et de les appliquer sur n'importe quel modèle multi-agents. La plateforme FPMAS permet d'abstraire ces problèmes de synchronisation et de distribution pour l'utilisateur, afin de simplifier au maximum la parallélisation des modèles. Comme il suffit de changer un paramètre pour appliquer n'importe quel mode de synchronisation à tous les modèles, il est très simple de comparer les performances et l'impact sur les résultats même pour un utilisateur avec des connaissances limitées en parallélisme.

La suite nos travaux s'orientent vers une analyse de nombreux modèles en vue d'identifier les besoins en synchronisation de différents types de modèles. De nouveau

modes seront éventuellement définis et implémentés dans la plateforme FPMAS, afin d'assurer une distribution simple et efficace pour une large variété de modèles.

Bibliographie

[1] P. Breugnot, B. Herrmann, C. Lang & L. Philippe, « 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)*, 2021, p. 54-61.

[2] B. Brock, A. Buluç & K. Yelick, « BCL: A Cross-Platform Distributed Data Structures Library », in *Proceedings of the 48th International Conference on Parallel Processing* (New York, NY, USA), ICPP 2019, Association for Computing Machinery, 2019, p. 1-10.

[3] K. Chandy & J. Misra, « Distributed Simulation: A Case Study in Design and Verification of Distributed Programs », *IEEE Transactions on Software Engineering* **SE-5** (1979), n° 5, p. 440-452.

[4] L. S. Chin, D. J. Worth, C. Greenough, S. Coakley, M. Holcombe & M. Kiran, « FLAME: An Approach to the Parallelisation of Agent-Based Applications », Tech. Report RAL-TR-2012-013, Rutherford Appleton Laboratory Technical Reports, 2012.

[5] N. Collier & M. North, « Parallel Agent-Based Simulation with Repast for High Performance Computing », *SIMULATION* **89** (2012), n° 10, p. 1215-1235.

[6] N. Collier, J. Ozik & C. M. Macal, « Large-Scale Agent-Based Modeling with Repast HPC: A Case Study in Parallelizing an Agent-Based Model », in *Euro-Par 2015: Parallel Processing Workshops* (Cham) (S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. Gómez Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer & M. Alexander, éds.), Lecture Notes in Computer Science, Springer International Publishing, 2015, p. 454-465.

[7] G. Cordasco, R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano & C. Spagnuolo, « A Framework for Distributing Agent-Based Simulations », in *Euro-Par 2011: Parallel Processing Workshops* (Berlin, Heidelberg) (M. Alexander, P. D'Ambra, A. Belloum, G. Bosilca, M. Cannataro, M. Danelutto, B. Di Martino, M. Gerndt, E. Jeannot, R. Namyst, J. Roman, S. L. Scott, J. L. Traff, G. Vallée & J. Weidendorfer, éds.), Lecture Notes in Computer Science, Springer, 2012, p. 460-470.

[8] E. W. Dijkstra, W. H. J. Feijen & A. J. M. van Gasteren, « Derivation of a Termination Detection Algorithm for Distributed Computations », in *Control Flow and Data Flow: Concepts of Distributed Programming* (Berlin, Heidelberg) (M. Broy, éd.), Springer Study Edition, Springer, 1986, p. 507-512.

[9] J. Ferber, *Les Systèmes Multi-Agents : Vers Une Intelligence Collective*, IIA Informatique, Intelligence Artificielle, InterEditions, 1995.

[10] J. Ferber & J.-P. Müller, « Influences and Reaction: A Model of Situated Multiagent Systems », *Proceedings of second international conference on multi-agent systems (ICMAS-96)* (1996), p. 72-79.

[11] O. Gutknecht & J. Ferber, « MadKit: A Generic Multi-Agent Platform », in *Proceedings of the Fourth International Conference on Autonomous Agents - AGENTS '00* (Barcelona, Spain), ACM Press, 2000, p. 78-79.

[12] D. R. Jefferson, « Virtual Time », *ACM Transactions on Programming Languages and Systems* **7** (1985), n° 3, p. 404-425.

[13] A. Malinowski & P. Czarnul, « Multi-Agent Large-Scale Parallel Crowd Simulation with NVRAM-based Distributed Cache », *Journal of Computational Science* **33** (2019), p. 83-94.

[14] P. Mathieu & Y. Secq, « Environment updating and agent scheduling policies in agent-based simulators », in *Proceedings of the 4th International Conference on Agents and Artificial Intelligence*, vol. 1, SciTePress, 2012, p. 170-175.

[15] D. Pawlaszczyk & S. Strassburger, « Scalability in Distributed Simulations of Agent-Based Models », in *Proceedings of the 2009 Winter Simulation Conference (WSC)*, 2009, p. 1189-1200.

[16] K. Popov, M. Rafea, F. Holmgren, P. Brand, V. Vlassov & S. Haridi, « Parallel Agent-Based Simulation on a Cluster of Workstations », *Parallel Processing Letters* **13** (2003), n° 04, p. 629-641.

[17] D. M. Rao & A. Chernyakhovsky, « Parallel Simulation of the Global Epidemiology of Avian Influenza », in *2008 Winter Simulation Conference*, 2008, p. 1583-1591.

[18] A. Rousset, B. Herrmann, C. Lang & L. Philippe, « A Survey on Parallel and Distributed Multi-Agent Systems for High Performance Computing Simulations », *Computer Science Review* **22** (2016), p. 27-46.

[19] A. Rousset, B. Herrmann, C. Lang, L. Philippe & H. Bride, « Nested Graphs: A Model to Efficiently Distribute Multi-Agent Systems on HPC Clusters », *Concurrency and ComputationPractice and Experience* **30** (2018), nº 7, p. e4407.

[20] X. Rubio-Campillo, « Pandora: A Versatile Agent-Based Modelling Platform for Social Simulation », in *Proceedings of SIMUL*, 2014, p. 29-34.

[21] D. Scerri, A. Drogoul, S. Hickmott & L. Padgham, « An Architecture for Modular Distributed Simulation with Agent-Based Models », in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems* (Toronto, Canada), vol. 1, 2010, p. 541-548.

[22] V. Suryanarayanan, G. Theodoropoulos & M. Lees, « PDES-MAS: Distributed Simulation of Multi-agent Systems », *Procedia Computer Science* **18** (2013), p. 671-681.

[23] P. Taillandier, D.-A. Vo, E. Amouroux & A. Drogoul, « GAMA: A Simulation Platform That Integrates Geographical Information Data, Agent-Based Modeling and Multi-scale Control », in *Principles and Practice of Multi-Agent Systems* (Berlin, Heidelberg) (N. Desai, A. Liu & M. Winikoff, éds.), Lecture Notes in Computer Science, Springer, 2012, p. 242-258.

[24] S. Tisue & U. Wilensky, « NetLogo: Design and Implementation of a Multi-Agent Modeling Environment », in *Proceedings of Agent*, vol. 2004, Springer Cham, Switzerland, 2004, p. 7-9.

[25] G. Vigueras, J. M. Orduña, M. Lozano & Y. Jégou, « A Scalable Multiagent System Architecture for Interactive Applications », *Science of Computer Programming* **78** (2013), nº 6, p. 715-724.

[26] U. Wilensky, « NetLogo Wolf Sheep Predation Model », Center for Connected Learning and Computer-Based Modeling, 1997.

[27] ———, « NetLogo Flocking Model », Center for Connected Learning and Computer-Based Modeling, 1998.

[28] ———, « NetLogo Virus Model », Center for Connected Learning and Computer-Based Modeling, 1998.

[29] Y. Xu, W. Cai, H. Aydt, M. Lees & D. Zehe, « Relaxing Synchronization in Parallel Agent-Based Road Traffic Simulation », *ACM Transactions on Modeling and Computer Simulation* **27** (2017), nº 2, p. 1-24.

[30] J. A. Yorke, N. Nathanson, G. Pianigiani & J. Martin, « Seasonality and the Requirements for Perpetuation and Eradication of Viruses in Populations », *American journal of epidemiology* **109** (1979), nº 2, p. 103-123.

Abstract. — Among simulation or modelisation methods, multi-agent systems are interesting candidate to simulate complex systems. When the size of the models increases, the use of parallel multi-agent systems is mandatory but comes with many issues. In this article, we are interested in the impact of synchronization on model implementation and on their execution. We highlight synchronization problems through model instances then we experimentally analyze the impact of synchronization on large scale testcases. To address the issues highlighted by this analyze, we propose a generic synchronization interface and its implementation in the FPMAS platform.

Keywords. — Multi-agent simulation, Parallelism, High Performance Computing, Synchronization.

Manuscrit reçu le 15 juillet 2021, révisé le 31 janvier 2022, accepté le 23 février 2022.