Modèles de distribution pour la simulation de trafic multi-agent

Matthieu Mastio

UNIVERSITÉ PARIS-EST

École Doctorale MSTIC

Institut français des sciences et technologies des transports, de l'aménagement des réseaux.

Thèse

Présentée pour l'obtention du grade de DOCTEUR
DE L'UNIVERSITE PARIS-EST
par

Matthieu Mastio

Modèles de distribution pour la simulation de trafic multi-agent

---

Spécialité : Informatique

Soutenue le 12 juillet 2017 devant un jury composé de :

Rapporteur	**Pr René Mandiau**	(Université de Valenciennes)
Rapporteur	**Pr Philippe Mathieu**	(Université de Lille I)
Président	**Pr Salima Hassas**	(Université de Lyon I)
Examinateur	**Dr Jean-Paul Jamont**	(Université Grenoble Alpes)
Directeur de thèse	**Dr Gérard Scémama**	(Université Paris Est, IFSTTAR)
Co-directeur de thèse	**Pr Omer Rana**	(Cardiff University)
Encadrant	**Dr Mahdi Zargayouna**	(Université Paris Est, IFSTTAR)

Thèse effectuée au sein de **L'Institut français des sciences et technologies des transports, de l'aménagement et des réseaux**

14-20 Boulevard Newton

77420 Champs-sur-Marne

France

Résumé

L'analyse et la prévision du comportement des réseaux de transport sont aujourd'hui des éléments cruciaux pour la mise en place de politiques de gestion territoriale. La simulation informatique du trafic routier est un outil puissant permettant de tester des stratégies de gestion avant de les déployer dans un contexte opérationnel. La simulation du trafic à l'échelle d'un ville requiert cependant une puissance de calcul très importante, dépassant les capacités d'un seul ordinateur.

Cette thèse est consacrée à l'étude de méthodes permettant l'exécution de simulations de trafic multi-agent large échelle. Nous proposons des solutions permettant de distribuer l'exécution de telles simulations sur un grand nombre de cœurs de calcul. L'une d'elle distribue directement les agents sur les cœurs disponibles, tandis que la seconde découpe l'environnement sur lequel les agents évoluent. Les méthodes de partitionnement de graphes sont étudiées à cet effet, et nous proposons une procédure de partitionnement spécialement adaptée à la simulation de trafic multi-agent. Un algorithme d'équilibrage de charge dynamique est également développé, afin d'optimiser les performances de la distribution de la simulation microscopique.

Les solutions proposées ont été éprouvées sur un réseau réel représentant la zone de Paris-Saclay. Ces solutions sont génériques et peuvent être appliquées sur la plupart des simulateurs existants. Les résultats montrent que la distribution des agents améliore grandement les performances d'une simulation de type macroscopique, tandis que le découpage de l'environnement se révèle plus adapté aux simulations de type microscopique. L'algorithme d'équilibrage de charge améliore en outre significativement l'efficacité de la distribution de l'environnement.

Mots-clé:

calcul distribué, multi-agent, simulation de trafic, partition de graphe, équilibrage de charge dynamique

Distributed models for multi-agent traffic simulation

Abstract

Nowadays, analysis and prediction of transport network behavior are crucial elements for the implementation of territorial management policies. Computer simulation of road traffic is a powerful tool for testing management strategies before deploying them in an operational context. Simulation of city-wide traffic requires significant computing power exceeding the capacity of a single computer.

This thesis studies the methods to perform large-scale multi-agent traffic simulations. We propose solutions allowing the distribution of such simulations on a large amount of computing cores. One of them distributes the agents directly on the available cores, while the second splits the environment on which the agents evolve. Graph partitioning methods are studied for this purpose, and we propose a partitioning procedure specially adapted to the multi-agent traffic simulation. A dynamic load balancing algorithm is also developed to optimize the performance of the microscopic simulation distribution.

The proposed solutions have been tested on a real network representing the Paris-Saclay area. These solutions are generic and can be applied to most existing simulators. The results show that the distribution of the agents greatly improves the performance of the macroscopic simulation, whereas the environment distribution is more suited to microscopic simulation. Our load balancing algorithm also significantly improves the efficiency of the environment based distribution.

Keywords:

distributed computing, multi-agent, traffic simulation, graph partitioning, dynamic load balancing

Remerciements

La réalisation d'une thèse de doctorat est une expérience de vie toute particulière, aussi éprouvante qu'enrichissante. Je souhaite réserver quelques mots de remerciement à destination des personnes qui m'ont soutenu dans cette aventure.

Pour commencer, bien sûr, je voudrais remercier mes encadrants pour tout le temps qu'ils m'ont consacré. Ce fut un grand plaisir de travailler sous leur direction. Merci à Gérard Scemama, pour ses conseils avisés, qui ont donné un cap à mes recherches. Sa grande expérience du monde de la recherche m'a permis de prendre de la hauteur par rapport à mes travaux, et de ne pas m'égarer dans mes décisions. Merci à Mahdi Zargayouna, pour son soutien de chaque instant. Il m'a donné une grande liberté d'action pendant cette thèse, et m'a toujours soutenu dans mes choix. Ses mots d'encouragement dans les moments de doute, ainsi que son support indéfectible ont été des facteurs déterminants à l'aboutissement de ce travail.

Un grand merci également à Omer Rana, qui m'a accueilli pour un séjour scientifique de trois mois à l'université de Cardiff. Il a tout fait pour faciliter mon installation, et a été d'une grande aide pour orienter mes recherches. Il m'a notamment donné l'accès au cluster de l'université de Cardiff afin d'y effectuer mes calculs, ce qui a été une étape cruciale dans l'avancée de mes travaux. Merci à l'action COST et à l'école doctorale MSTIC d'avoir financé ce séjour, qui fut une expérience très enrichissante, aussi bien sur le plan scientifique que personnel.

Je tiens aussi évidemment à remercier les membres de mon jury de thèse qui m'ont fait l'honneur d'évaluer mon travail. Je remercie Philippe Mathieu et René Mandiau, pour le temps et l'énergie consacrés à rapporter ce mémoire, mais aussi Salima Hassas et Jean Paul Jamont, pour avoir accepté de faire parti de mon jury d'évaluation. Je les remercie pour leurs remarques pertinentes, qui ont grandement contribué à améliorer ce mémoire, ainsi que pour la discussion constructive qui a suivi ma soutenance.

Pour le support administratif qu'elles m'ont apporté, toujours avec le sourire, je voudrais remercier Joëlle Guillot et Annie Thuilot du secrétariat du GRETTIA, Nathalie Galéa du secrétariat général de l'IFSTTAR ainsi que Sylvie Cach du secrétariat de l'Ecole Doctorale MSTIC. Un merci tout particulier, également, à Mustapha Tendjaoui, pour son aide technique, sur tous types de problèmes.

Bien sûr, cette liste ne serait jamais complète sans y mentionner mes collègues, qui ont égayé chacune de mes journées à l'institut. Je les remercie pour leur bonne humeur, ainsi que les discussions passionnantes et les débats mouvementés des pauses de midi (ou des pauses surprises au cours de la journée). Certains sont devenus des amis (et j'espère de tout coeur qu'ils le resteront) avec qui j'ai partagé des moments inoubliables. La liste n'est bien sûr pas exhaustive, mais voici quelques noms qui je sais occuperons une place particulière dans ma mémoire : Dihya, Johanna, Rony, Guillaume, Moncef, Anne-sarah, Hani, Remi, Josquin, Arthur, Mahdi, Fairouz, Florian, Latifa, Mustapha, Mohamed, Florence, Farida,

Pierre, Nathan, Adrien, Adeline, Etienne, Xavier, Asma, Régine, Manuel, Cyril.

Mes plus tendres pensées pour ma famille, qui a toujours cru en moi, que ça soit dans la réalisation de cette thèse ou dans mes choix de vie en général. Un merci tout particulier à mon grand-père, pour sa relecture attentive, ainsi que pour la belle surprise qu'il m'a faite le jour de ma soutenance.

Je voudrais enfin réserver une place à ceux de mes proches qui ont joué un rôle particulier dans cette expérience. Merci à Matthis, qui fut l'instigateur de cette thèse, merci à tous les colocs que j'ai eu la chance de côtoyer pendant ces trois années, merci à Laura, Cyril, Angel, Lulu, JM, Neff, Gabi, Paula, Moncef, Dihya, Arthur, Sauk, Élise, ainsi qu'à Conrad et Guigui. Merci à Ariadna, pour l'amour et le courage qu'elle m'a insufflé, particulièrement essentiels lors de la dernière ligne droite. Merci à vous tous pour votre grand soutien, pour les couleurs que vous avez apporté à ma vie parisienne.

Table des matières

Liste des figures

Introduction

Motivations

L'optimisation des déplacements de personnes et de véhicules au sein des infrastructures routières d'une ville limite les pertes de temps et d'énergie, et contribue ainsi à l'ensemble de la société. La modélisation du trafic routier est une discipline qui a débuté dans les années 1950 avec les travaux de Lightill, Witham et Richards [53], qui ont été les premiers à modéliser le trafic à l'aide d'équations d'écoulement de fluide. Ce type d'approche a longtemps été privilégiée dans les travaux de recherche qui ont suivi. Puis, dans les années 2000, les capacités des systèmes informatiques se sont rapidement améliorées. Il est devenu possible de représenter un très grand nombre de phénomènes simples, puis de les combiner afin d'obtenir une simulation numérique capable de décrire des phénomènes d'une complexité élevée, tel que l'évolution du trafic routier.

Le développement de simulations de mobilité est pertinent dans plusieurs contextes, et permet la réalisation de plusieurs objectifs. La simulation peut servir à valider l'impact de l'usage de systèmes coopératifs [37], à tester les modifications de comportements suite à l'introduction de nouveaux services de mobilité tels que le covoiturage, les systèmes de guidage GPS dynamiques, ou encore le partage de véhicules. L'intérêt de l'utilisation d'un simulateur est de pouvoir définir et tester de nombreux scénarios, sans impacter le trafic réel [1, 40].

Cependant, les systèmes de transport deviennent progressivement plus complexes, intégrant de plus en plus d'entités connectées (appareils mobiles, véhicules connectés, etc). Il devient vital que les outils de simulations prennent en compte cet état de fait. En effet, avec la généralisation de l'information temps-réel des voyageurs, le comportement des réseaux de transport modernes devient de plus en plus difficile à prévoir. Pour ces raisons, la simulation multi-agent, qui adopte une approche centrée sur l'individu, est l'un des paradigmes les plus prometteurs pour la conception et la réalisation de telles applications.

Une simulation de trafic multi-agent simule le comportement de voyageurs interagissant dans un environnement complexe, dynamique et ouvert, duquel ils ont une perception partielle [5]. Dans ce genre de simulation, il est important de modéliser et de simuler un

nombre réaliste de voyageurs pour observer correctement les effets des décisions individuelles. Le projet Européen *Instant Mobility*[1] par exemple, a pour objectif d'alimenter une plate-forme de guidage avec des requêtes de déplacements et des positions dynamiques des voyageurs et des véhicules. Pour permettre la démonstration des capacités de la plate-forme dans un contexte opérationnel, l'IFSTTAR a implémenté SM4T [86](*Simulator for Multiagent MultiModal Mobility of Travelers*), un simulateur dont la finalité est de pouvoir s'exécuter avec un volume réel de voyageurs.

Problématique

La simulation d'un nombre réel de voyageurs à l'échelle d'une grande ville (pouvant aller jusqu'à plusieurs millions de voyageurs) présente le désavantage d'être extrêmement gourmande en ressources de calcul, et était considérée comme impossible jusqu'à récemment.

L'émergence d'architectures permettant la distribution des traitements sur un grand nombre d'hôtes à en effet ouvert de nouvelles opportunités. La version actuelle de SM4T, comme la majorité des simulateurs de mobilité de voyageurs actuels, ne permet pourtant pas encore une distribution sur de tels systèmes.

Cela induit une limite quant au nombre de voyageurs, de moyens de transports et à la taille des réseaux considérés. Par conséquent, la prévision des effets des réglementations et des stratégies d'information sur de grands réseaux en présence de voyageurs connectés et informés en temps réel devient très difficile. Notre principal objectif est donc de tester le passage à l'échelle de ce type de simulateurs. Nous désirons fournir des patrons de distribution génériques et reproductibles, qui pourraient être utilisés par tout simulateur multi-agent de mobilité, ou encore plus généralement par tout simulateur d'agents situés.

Contributions

Les contributions de cette thèse se divisent en trois axes.

Tout d'abord, nous définissons un simulateur de trafic représentatif, constituant un cadre de travail pour nos expérimentations. Par souci de généricité, il est aussi simple que possible. Les solutions particulières pourront considérer des simulateurs plus complexes, sans remettre en cause les résultats que nous présentons dans ce mémoire. Il est pour autant aussi complexe que nécessaire, incluant deux modules permettant de simuler le déplacement dans un contexte microscopique ou macroscopique. Il est ainsi représentatif de la très grande majorité des simulateur existant dans la littérature. Nous n'avons gardé dans ce simulateur que les aspects qui peuvent avoir un impact sur la problématique de distribution.

[1] http://www.instant-mobility.com/

Nous proposons ensuite deux méthodes qui permettent effectivement de déployer le simulateur sur un nombre illimité d'hôtes. La première distribue les agents représentant les utilisateurs du réseau routier sur les hôtes, tandis que la deuxième découpe le réseau lui-même en plusieurs parties et distribue ces parties entre les hôtes. Nous appliquons ces méthodes sur nos deux modules et fournissons une critique de leur efficacité en fonction du contexte.

Finalement, la troisième contribution porte sur l'équilibrage de charge de la simulation microscopique. Cet équilibrage est nécessaire afin de fournir des performances satisfaisantes lors de la distribution d'un système évoluant dynamiquement. Nous proposons un algorithme permettant de diffuser la charge entre les hôtes pendant l'exécution de la simulation.

Organisation du mémoire

Ce mémoire est divisé en deux parties. La première présente l'état de l'art des deux domaines sur lesquels nous travaillons : la simulation de trafic et le calcul parallèle. Un chapitre sera consacré à chacun de ces domaines. La deuxième partie décrit nos contributions. Elle est composée de trois chapitre, présentant chacun l'un des axes décrits dans la partie précédente.

Première partie. État de l'art.

Cette partie positionne notre travail par rapport à ceux réalisés antérieurement.

Chapitre 1. La simulation.

Le chapitre 1 explicite la notion de simulation. Nous y introduisons le paradigme multi-agent, puis proposons un état de l'art des simulateurs de trafic existants dans la littérature.

Chapitre 2. Le calcul haute performance.

Le chapitre 2 présente les architectures parallèles existantes, ainsi que leurs limitations. Nous y décrivons précisément les modèles à mémoire partagée et à mémoire distribuée. Ce chapitre fournit également un état de l'art de plateformes multi-agent distribuées existantes.

Deuxième partie. Contributions.

Nous décrivons dans cette partie les différentes étapes qui ont été nécessaires à la mise en place de solutions performantes permettant de distribuer une simulation de trafic multi-agent sur plusieurs hôtes.

Chapitre 3. Simulateur de déplacement.

Le chapitre 3 décrit le simulateur de trafic expérimental que nous avons développé. Nous introduisons ici les concepts mathématiques de la théorie des graphes, puis décrivons en détail le comportement des agents dans les différents modèles proposés.

Chapitre 4. Méthodes de distribution.

Dans le chapitre 4, nous décrivons le problème de la distribution d'une simulation de trafic de manière formelle. Nous présentons les techniques les plus efficaces pour le découpage de graphes, et proposons un algorithme de découpage spécifique à notre simulateur. Nous exposons ensuite les détails d'implémentation de deux méthodes de distribution de la simulation. Enfin, nous montrons les résultats obtenus grâce à ces méthodes.

Chapitre 5. Diffusion dynamique de charge.

Le chapitre 5 présente la problématique de l'équilibrage de charge dynamique d'un simulateur de trafic distribué. Nous étudions dans ce chapitre différentes méthodes d'équilibrage de charge dynamique, puis proposons un algorithme spécifiquement adapté à notre problème. Nous analysons enfin les performances de cet algorithme dans un contexte expérimental.

Partie I

Etat de l'art

Chapitre 1

La simulation de trafic

Sommaire

1.1 Introduction

Le transport efficace de biens et de personnes au travers d'un réseau routier est un problème passionnant. Le trafic routier est un phénomène complexe pour de multiples raisons. Tout d'abord, il implique un nombre élevé d'acteurs situés dans de larges zones géographiques, dont le nombre évolue dynamiquement avec le temps. Plus il y a d'usagers dans le système, et plus le nombre d'interactions sera important. La complexité du système est donc inégale et aléatoire. D'autre part les objectifs des utilisateurs ne sont pas forcément compatibles et un grand nombre de facteurs ne sont pas contrôlables, ni par les opérateurs, ni par les usagers, comme par exemple l'état de la route et la météo.

Malgré ces difficultés, le contrôle, l'analyse et l'optimisation du réseau routier sont cruciaux. Un réseau routier laissé sans surveillance aurait des conséquences importantes sur l'économie nationale. Le phénomène qui concerne le plus directement les usagers est bien sûr la formation de bouchons (une congestion se forme lorsque la demande en un point donné dépasse sa capacité), mais de nombreux autres facteurs sont aussi pris en compte. Pour les usagers, il est important d'optimiser leurs temps de parcours. Des systèmes de guidage leur sont désormais disponibles, qui leur proposent des itinéraires dynamiquement en fonction de l'état du trafic. La problématique environnementale est également particulièrement importante actuellement. Il est possible grâce à une gestion efficace du réseau de limiter l'émission de gaz nocifs. La gestion de l'usure des infrastructures est aussi un élément essentiel pour le bon fonctionnement du système.

De nombreuses stratégies sont proposées dans la littérature scientifique pour optimiser ces facteurs. Cependant, leurs implémentations dans un contexte opérationnel produit parfois des résultats inattendus, voir contre-productifs. Les solutions intuitives ne sont pas toujours les plus efficaces. Un exemple bien connu est appelé le paradoxe de Braess [15]. Soit la configuration suivante : initialement, deux itinéraires permettent de se rendre d'un point A à un point B. Ces itinéraires comportent deux sections, l'une avec un temps de parcours fixe et l'autre avec un temps de parcours dépendant du nombre d'usagers sur la portion (figure 1.1). Une nouvelle route rapide (instantanée dans l'exemple) est ensuite construite reliant ces itinéraires (figure 1.2). L'intuition nous laisserait penser que l'ajout de cette route permettrait d'améliorer le temps de parcours des usagers.

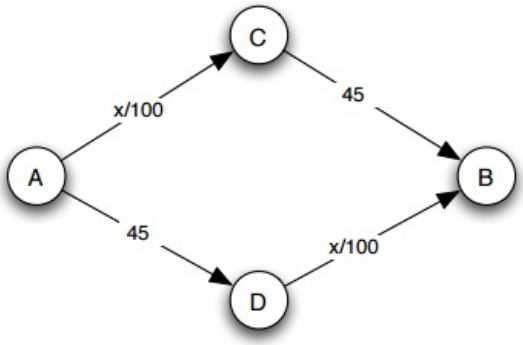

Figure 1.1: Une configuration de Braess

Mais Braess prouve que le système atteint une situation d'équilibre (où aucun voyageur ne gagnerait à changer d'itinéraire) dans laquelle le temps de parcours global des voyageurs est supérieur à celui du système initial. Ceci est dû à la nature égoïste des voyageurs, qui cherchent avant tout à maximiser leur bien être personnel plutôt que le bien être global. Certains aménagements du réseau qui ne prendraient pas en compte cette attitude indi-

vidualiste pourraient alors se révéler contre-producifs. Ainsi, par exemple, les conditions de trafic se sont améliorées à Séoul après la destruction d'une voie rapide [31].

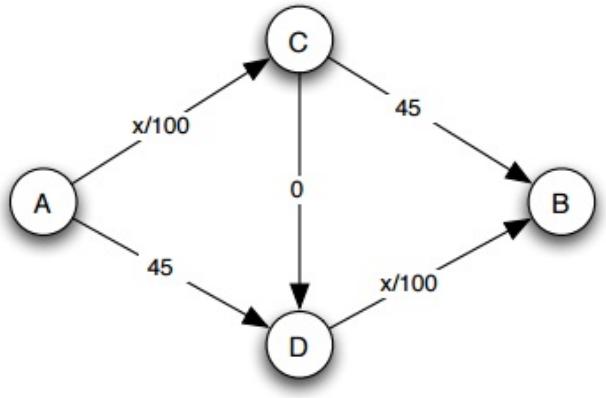

Figure 1.2: La même configuration, après l'ajout de la voie rapide

La situation décrite par Braess est une abstraction simple que l'on peut décrire facilement de manière analytique. Mais de nombreux problèmes relatifs au trafic routier dépassent le champs de la méthode analytique, de part leur trop grande complexité. Les réseaux sont dans la réalité bien plus étendus que dans le problème de Braess, et les conditions de circulation difficilement quantifiables numériquement. C'est pour cette raison que la simulation de scénarios routiers s'avère si précieuse, pour tester les stratégies de gestion permettant de répartir efficacement la demande et d'assouplir les contraintes exercées sur le réseau.

Une simulation de trafic routier est une version informatique d'un modèle qui est exécuté de manière itérative afin d'étudier les implications des éléments définis. Elle a pour but de représenter le réseau routier ainsi que ses usagers et répond aux besoins des chercheurs, des agences gouvernementales et des cabinets de conseil privés de tester, évaluer et démontrer un plan d'action en étude avant son implémentation réelle.

1.2 Le paradigme multi-agent

Le paradigme multi-agent est un objet d'étude qui a débuté avec la recherche sur l'intelligence artificielle distribuée, une sous-branche de l'intelligence artificielle. Il s'agit d'un paradigme qui propose une alternative aux méthodes analytiques utilisée jusqu'ici. Au lieu de chercher a décrire la réalité dans sa globalité par des équations, cette méthode s'intéresse au comportement de petites entités (les agents).

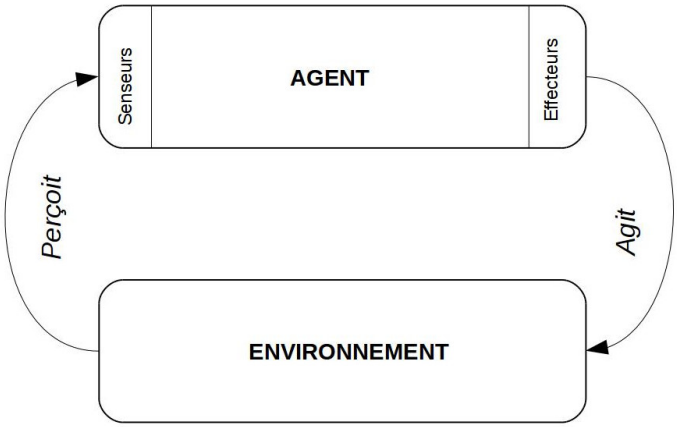

Figure 1.3: Un agent

1.2.1 Définition

La notion d'agent, bien que ne disposant pas d'une définition universelle, possède un certain nombre de caractéristiques qui font consensus dans le monde de la recherche. Ferber [32] en donne une définition assez générale :

Un agent est une entité réelle ou virtuelle, évoluant dans un environnement, capable de le percevoir et d'agir dessus (figure 1.3), qui peut communiquer avec d'autres agents, qui exhibe un comportement autonome, lequel peut être vu comme la conséquence de ses connaissances, de ses interactions avec d'autres agents et des buts qu'il poursuit.

Le mode d'action d'un agent a été formalisé par Bratman [16] dans son framework BDI : chaque agent possède une base de connaissances (Belief), va mettre en place un ensemble d'actions (Intention) en vue de réaliser un objectif (Desire). Les agents disposent généralement des caractéristique suivantes [42] :

1. Autonomie : un agent agit sans intervention humaine et contrôle ses actions et ses états internes.
2. Réactivité : un agent réagit rapidement aux changements intervenus dans son environnement.
3. Proactivité : un agent peut prendre des décisions en anticipant ce qui peut se produire dans son environnement.
4. Sociabilité : un agent peut interagir avec d'autres agents si il le juge nécessaire.

La caractéristique la plus importante d'un agent est sa capacité à agir de manière autonome, c'est à dire sans intervention extérieure lui donnant la marche à suivre en fonction

des situations qu'il pourrait rencontrer. Si les actions d'un agent découlent mécaniquement des perceptions qu'il a de son environnement (comme c'est le cas avec les automates cellulaires), il est appelé agent réactif. Si son comportement est plus élaboré, avec par exemple un processus d'apprentissage grâce auquel il va affiner ses actions en tenant en compte du résultat de ses actions précédentes, il est dénommé agent cognitif. Un agent cognitif sera généralement plus complexe, et nécessitera plus de temps de calcul qu'un agent purement réactif.

En outre, la caractéristique de sociabilité d'un agent nous mène à la définition d'un systèmes multi-agent (SMA) : il s'agit d'un programme informatique, décrivant un environnement virtuel, dans lequel sont placés plusieurs agents. Ces agents vont agir et interagir entre eux et avec leur environnement. Un système multi-agent est caractérisé comme suit [80] :

1. Chaque agent dispose d'une connaissance incomplète du problème.
2. Il n'y a pas de système de contrôle global.
3. Les données sont décentralisées.
4. Les calculs sont asynchrones.

1.2.2 Topologie du système

Dans un SMA, seule l'information locale est disponible pour les agents. Il n'y a pas d'autorité centrale diffusant l'information aux agents de manière artificielle, ou optimisant l'efficacité du système en imposant des actions aux agents. La seule information disponible pour les agents est soit celle qu'il vont directement percevoir, soit l'information qui leur sera communiquée par d'autres agents. C'est pourquoi les modes d'interaction entre les agents sont au centre du problème lors de la conception de SMA. Est ce que chaque agent va communiquer avec tous les autres agents présents dans le système ou seulement avec un sous-ensemble ? Comment définir ce sous-ensemble, et va t'il évoluer avec le temps ? Par exemple, un agent pourra interagir avec ses voisins géographiques les plus proches, ou alors avec d'autres agents partageant les mêmes centres d'intérêt que lui dans un réseau social.

La nature des interactions entre les agents dans un modèle est appelée sa topologie. Macal [55] décrit les topologies fréquemment rencontrées dans les SMA (figure 1.4). Le modèle soupe (a), ou aspatial, décrit une situation dans lequel l'emplacement des agents n'est pas significatif pour le modèle. L'automate cellulaire (b) est un mode d'interaction dans lequel l'espace est découpé en cellules. Chaque agent se déplace de cellule en cellule et communique avec les agents situés dans les cellules voisines. Dans un espace euclidien (c), les agents se déplacent cette fois librement dans un espace continu, qui n'est plus découpé

en cellule. La topologie en réseau (d) décrit les interactions entre les agents de manière plus générale, puisque des agents peuvent interagir sans être proches physiquement. Les systèmes d'information géographique (e) permettent quand à eux de faire évoluer les agents dans des espaces géographiques réalistes. Cette liste de topologies n'est pas exhaustive, et il est en outre fréquent de trouver des modèles combinant plusieurs types topologiques.

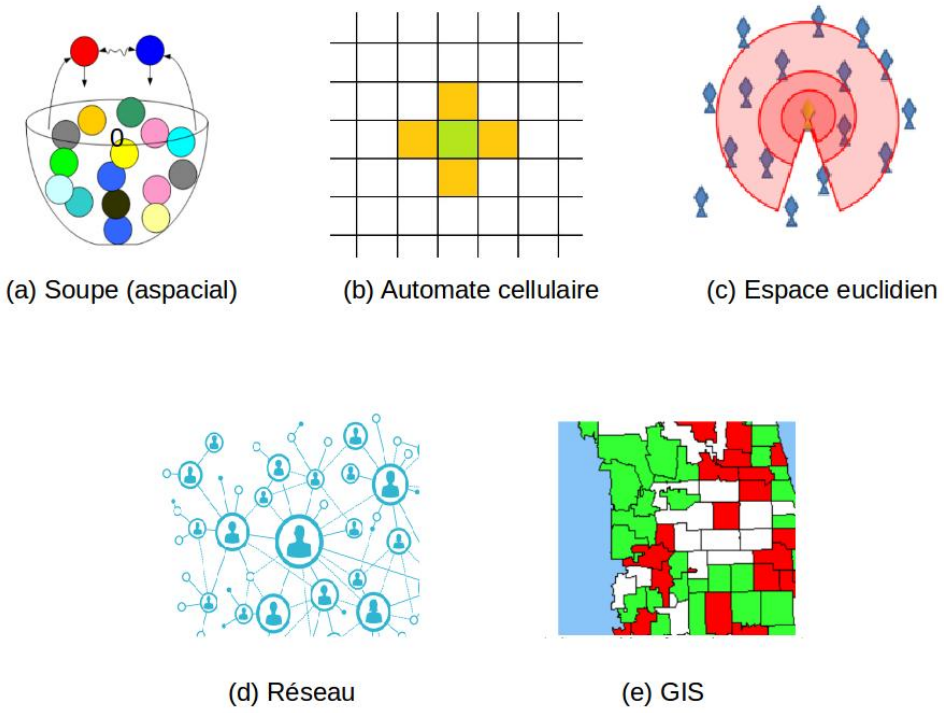

Figure 1.4: Topologies des interactions entre les agents

1.2.3 La gestion du temps et de l'aléa

Nous avons défini la simulation multi-agent comme étant un ensemble d'entités (les agents) interagissant selon un patron de communication. Cependant, pour que le système soit cohérent, il est indispensable d'organiser les actions des agents dans le temps. Par exemple, un agent ne pourra pas réagir à un instant t à une action effectuée à l'instant $t+1$ par un autre agent. Les simulateurs existant se placent en deux grandes catégories en ce qui concerne la gestion du temps : la simulation à évènements discrets et la simulation continue.

Dans la simulation à évènements discrets (*event-driven*), la simulation est mise à jour

chaque fois qu'un évènement discret se produit. La simulation est ainsi une suite d'évènements qui vont modifier l'état du système. Celui-ci ne change pas entre les évènements, et il est donc possible d'avancer dans le temps d'événement en évènement. Ce procédé, très efficace pour les systèmes de petite taille ou dont les agents ne changent pas souvent d'état, se révèle compliqué à utiliser avec des systèmes d'une taille importante qui comportent de nombreux agents changeant d'état fréquemment.

Dans la simulation continue (time-driven), le temps est découpé en intervalles réguliers et on examine l'état du système à chaque pas de temps (δt). Les agents mettent également à jour leur base de connaissances du monde et agissent en conséquence pour réaliser leur objectif. Le temps est la variable indépendante dans le modèle et la granularité d'observation est choisie par le programmeur. Plus celle-ci sera fine et plus la simulation sera précise et prendra du temps à s'exécuter. Cette méthode est appropriée pour les systèmes simulant un grand nombre d'agents qui changent fréquemment d'état.

D'autre part, après avoir défini le modèle temporel d'un système, il est nécessaire d'en gérer l'aléa. Si l'on veut reproduire la variabilité d'un système stochastique, il faudra utiliser des modèles statistiques qui feront, avec les même conditions initiales, varier le résultat de la simulation d'une exécution à une autre. On peut également choisir une exécution déterministe : tout est connu dès le début, avec aucun facteur aléatoire. C'est une manière d'évaluer précisément les impacts induits par un changement d'un paramètre du système, grâce à la suppression du bruit généré par l'aléatoire.

1.2.4 Le modèle de ségrégation, un exemple d'émergence

Le paradigme multi-agent est un outil conceptuel. On ne tente plus de décrire un système dans sa globalité, mais plutôt du point de vue des unités le constituant. L'ensemble des actions individuelles de ces unités vont influencer l'état général du système. Des évènements, parfois difficilement prévisibles grâce aux seules conditions initiales du système vont se produire au niveau macroscopique. Ces phénomènes sont appelés propriétés émergentes du système. L'observation et l'analyse des ces émergences sont au cœur de la philosophie multi-agent. Elles vont permettre de décrire le système de manière souvent plus précise que l'on ne pourrait le faire à l'aide d'équations.

Un exemple célèbre d'émergence macroscopique est obtenu grâce au modèle de ségrégation proposé par Schelling [76]. Son but était de démontrer que des quartiers résidentiels ségrégationnistes pouvait apparaître même si individuellement les habitants sont plutôt tolérants à la vie dans un environnement mixte. Pour modéliser le comportement d'un habitant, il part du postulat suivant : un habitant accepte de vivre avec un voisinage majoritairement différent, sauf si il y est trop minoritaire.

Pour tester son hypothèse, il propose de placer des agents sur une grille, à raison d'un

agent par cellule. Ces derniers n'interagissent qu'avec leurs huit voisins immédiats. Chaque agent accepte son voisinage si il y a au moins 37.5% de voisins semblables. Si un habitant est mécontent, il déménage en choisissant au hasard une cellule correspondant à ses désirs. Des configurations fortement homogènes vont alors apparaître (voir figure 1.5). Schelling exhibe ainsi une propriété globale du système qui émerge uniquement des interactions locales des agents composant ce système. Même des agents définis de manière simple peuvent donner des informations précieuses sur la dynamique du système qu'ils modélisent.

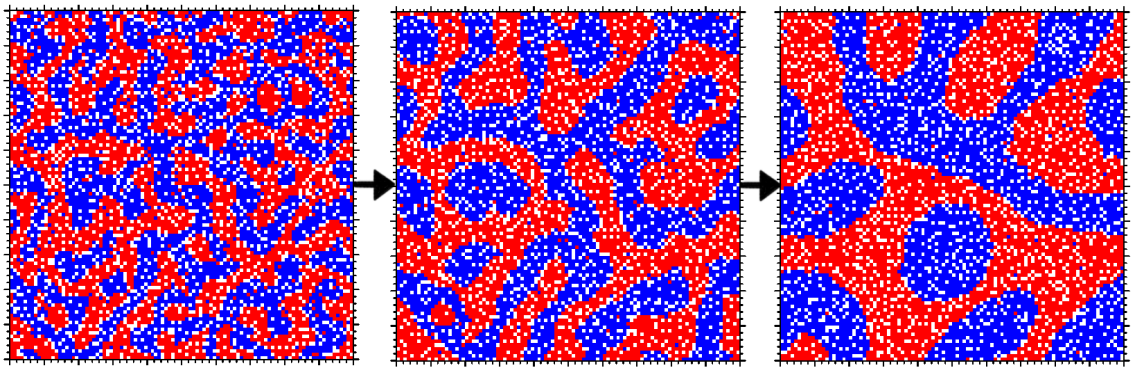

Figure 1.5: Un exemple d'exécution du modèle de Ségrégation défini par Schelling

1.2.5 Les systèmes multi-agents au service de la simulation

Les SMA (Systèmes Multi-Agents) sont nés du besoin de modéliser les comportements organisationnels des êtres humains à partir de leurs prises de décisions individuelles. En émulant leurs interactions sociales, on peut alors voir apparaître des structures sociales de plus haut niveau. Les simulations multi-agents essayent d'incorporer des comportements et des interactions humaines, et de mesurer l'impact de leurs activités.

Les SMA peuvent être utilisés pour expérimenter sur des systèmes qui sont trop difficiles à observer à cause de leur complexité ou de la lenteur d'évolution des phénomènes observés. Grâce à ces modèles, les expériences vont pouvoir être répétés avec plusieurs configurations, permettant de tester différents scénarios. Ils sont également un outil d'enseignement très efficace. Un système complexe peut être expliqué grâce à leur simulation, et des tests peuvent être effectués en guise de démonstration, exhibant les relations de cause à effet sous-jacentes au système. Ils peuvent ainsi être utilisés comme un outil d'aide à la décision, en facilitant, grâce à la démonstration, la communication nécessaire à la mise en place d'une politique.

La notion d'agent intelligent est considérée comme le paradigme de programmation le plus important depuis la conception orientée objet. Il a de nombreuses applications, dans des domaines aussi variés que la simulation d'écosystèmes, d'économies, dans le contrôle en

temps réel, dans la gestion de réseau, etc.

1.2.6 Plateformes multi-agents générales

La simulation d'un problème donné est très spécifique. C'est la raison pour laquelle il est impossible de proposer un outil de simulation multi-agent qui permettrait de modéliser directement n'importe quelle situation. Cependant, il existe un ensemble de caractéristiques partagées par la plupart des simulateurs. C'est pourquoi de nombreuses plateformes multi-agents à visée généraliste ont été proposées par les chercheurs. Ces plateformes ont pour but de faciliter le développement de nouveaux modèles, en prenant en charge les fonctions de base de tout simulateur, telles que la communication entre les agents, la gestion de leur cycle de vie, de leur perception, et de l'environnement en général. Ces plateformes fournissent en outre la plupart du temps une interface graphique permettant de représenter visuellement le déroulement de la simulation.

Parmi les plateformes les plus connues, on peut citer Mason [54], un simulateur à évènements discrets entièrement écrit en JAVA. Il a été conçu pour être utilisé dans un large champs d'application, grâce à sa grande extensibilité. Il sépare clairement l'aspect graphique du modèle, permettant de changer dynamiquement de représentation. Netlogo [84] est un environnement de programmation multi-agent à visée éducative, ou pour des experts souhaitant modéliser un phénomène sans connaissance préalable en informatique, en fournissant de nombreux modèles préprogrammés en guise d'exemple. Il utilise une version modifiée du langage de programmation logo. Repast Symphony [70] est une plateforme multi-agent très utilisée qui a de nombreuses implémentations dans différents langages. Elle propose des fonctionnalités avancées telles que la programmation génétique et la régression. Son système de projection permet de décrire efficacement des systèmes présentant des topologies diverses. Jade [13] est un framework entièrement en JAVA. Il simplifie l'implémentation de SMA grâce à un middleware qui respecte les spécifications FIPA [12]. Deux applications respectant ces spécifications peuvent interagir de manière transparente, même si elles ne sont pas développées dans le même langage. Les agents peuvent en outre être distribués entre plusieurs machines, ne possédant pas forcément le même système d'exploitation.

1.3 Les simulateurs de trafic

La simulation est devenue un élément incontournable de l'ingénierie de trafic, grâce aux avancées significatives des sciences de l'information, qui ont largement contribué à son développement. Les machines actuelles permettent en effet de simuler des zones étendues, d'intégrer des informations géographiques très précises (GIS[1]), ainsi que de mettre en place

[1] Geographical Information System

des démonstrations visuelles attrayantes. La simulation de trafic a permis l'élaboration de logiciels informatiques permettant l'aide à la décision, la création de nouveaux systèmes de transports et à l'amélioration générale de la qualité de service.

Il existe dans la littérature de nombreux outils de simulation qui répondent à toutes sortes de problématiques, pouvant aller de la prévision de congestions, à l'aide à la conception, ou encore à l'évaluation des impacts du trafic sur la qualité de l'air. Ces outils sont classés en fonction du niveau de détail dans la modélisation du flux de trafic. Chacun d'eux répondent à des besoins spécifiques, et on discrimine généralement deux échelles de simulation : l'échelle microscopique et l'échelle macroscopique. Une troisième échelle, appelée mésoscopique peut également être distinguée. Elle se situe a un niveau de détail entre le microscopique et le macroscopique et est généralement une hybridation de ces deux échelles principales. Certains systèmes sont spécialisés dans un domaine, d'autres sont capables de modéliser plusieurs échelles.

Étant donnée la puissance des machines d'aujourd'hui, la plupart des simulateurs récents sont des simulateurs à temps continu (time-driven). D'autre part, la nature du trafic routier est hautement aléatoire. C'est la raison pour laquelle les simulateurs sont généralement de nature stochastique. Il existe cependant des simulateurs déterministes, qui peuvent être vus comme une représentation d'un état de trafic moyen.

1.3.1 La simulation macroscopique

Il est parfois utile de simuler de grandes zones géographiques afin de disposer d'une vision large du système de trafic routier. Un simulateur capable de représenter de grandes zones peut être utile pour prévoir la charge sur le système, tester les politiques de gestion du trafic destinées à limiter les pressions exercées sur le système. Une application courante de la simulation macroscopique est l'analyse de l'exploitation sur de larges zones urbaines ou sur le réseau routier national.

A grande échelle, le comportement du trafic routier est similaire à l'écoulement d'un fluide dans un canal [53]. Dans ce type de simulation, les utilisateurs ne sont pas représentés individuellement. Un flot de véhicules est l'ensemble des véhicules parcourant une voie à une période donnée. Un véhicule situé à un moment donné en un point donné circulera à la vitesse du flot en ce point et à ce moment (figure 1.6). La simulation macroscopique utilise des notions telles que la vitesse moyenne, le débit et la densité pour décrire le flux d'usagers. Les conditions du trafic sont déterminées par la relation reliant ces notions. Cette agrégation permet d'économiser des ressources de calcul et des simulations d'un grand nombre d'utilisateurs sont rendues possibles.

Dagenzo [25] propose un modèle représentant le réseau routier comme un graphe dirigé où chaque nœud représente une intersection et chaque arc un segment de route. Pour

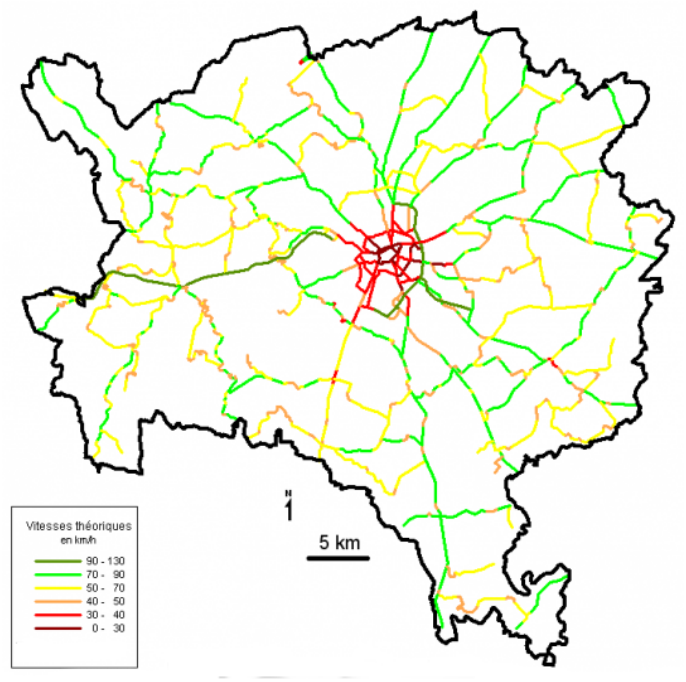

Figure 1.6: Simulation au niveau macroscopique

représenter le flux d'usagers naviguant dans le graphe, il utilise le diagramme fondamental du trafic routier [38]. Pour mesurer les flots de véhicules, on s'intéresse à des variables globales telles que le débit (le nombre de véhicules passant pendant une période en un point donné) ou la concentration (le nombre de véhicules par unité de longueur). La relation entre ces variables est décrite par le diagramme fondamental. Dans ce modèle, chaque portion de route possède en outre une densité critique à partir de laquelle le débit commence à diminuer, et, dans le pire des cas, des congestions commencent à se former. Il s'agit d'une approche conceptuelle qui nécessite un étalonnage grâce à des mesures réalisées sur le terrain [41].

Utilisant un modèle très similaire, METANET [62] est un outil de simulation macroscopique déterministe dans des réseaux de topologie arbitraire. Cette approche permet la simulation de tous types de conditions de trafic (trafic libre, dense, congestionné). Il permet également de prendre en compte des actions de contrôle d'un opérateur, tels que les affichages d'informations où le guidage par GPS.

Des modèles macroscopiques empruntant directement des notions de physique des fluides ont également été développés. Par exemple MASTER [39] utilise la théorie cinétique des gaz de Boltzmann pour mettre en équations le comportement macroscopique des véhicules.

1.3.2 La simulation microscopique

Aussi efficace que soit la simulation macroscopique, elle n'est pas adaptée pour toutes les situations. Il lui est impossible de décrire les conséquences de phénomènes impactant les utilisateurs individuels. L'aménagement d'un carrefour (remplacement d'un rond point par des feux de circulation), l'ajout d'un voie d'autoroute, ou encore la communication inter-véhiculaire (VANET) ne peuvent être étudiés à l'aide d'équations décrivant globalement les flux du trafic. Ce type de cas est le domaine de la simulation au niveau microscopique.

La simulation microscopique considère chacun des véhicules individuellement (figure 1.7). Durant la simulation, ces derniers se déplacent dans le réseau routier en s'adaptant continuellement à leur environnement. Ceci implique des interactions véhicule-environnement ainsi que de nombreuses interactions inter-véhicules (particulièrement au niveau urbain) qui sont modélisées de manière à reproduire le comportement réel des conducteurs.

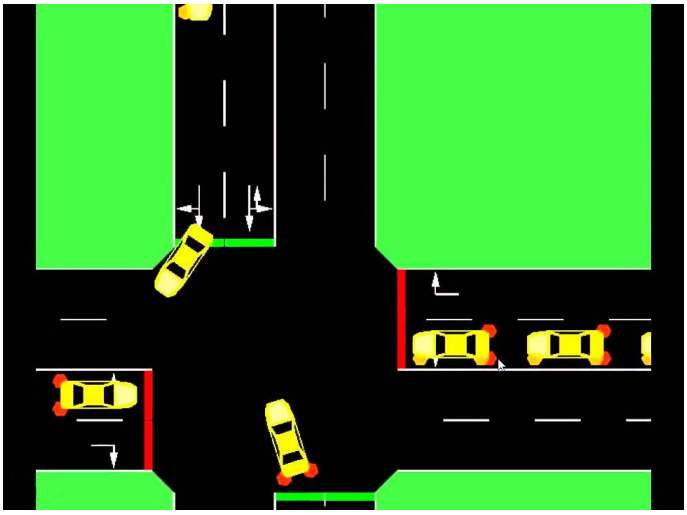

Figure 1.7: Simulation au niveau microscopique (obtenue avec SUMO)

Dans un système microscopique, le comportement de chaque véhicule dépend des autres véhicules qui l'entourent [56]. Les variables utilisées ici sont la position, la vitesse et l'accélération de chaque véhicule présent sur le réseau. Les véhicules vont calculer leurs vitesses comme le ferait un conducteur dans la réalité : en se fondant sur la vitesse du ou des véhicules qui le précèdent, et, éventuellement du véhicule qui le suit.

Ce niveau de détail élevé nécessite une puissance de calcul importante, et des simulations sur de grands réseaux peut se révéler très coûteuse en temps de calcul. C'est pourquoi la simulation microscopique est généralement utilisée pour des simulations sur des réseaux de tailles restreintes. La demande est communément représentée à l'aide d'une matrice origine-destination, qui indique le point de départ et l'endroit où se rend chaque utilisateur présent sur le réseau lors de l'intervalle de temps simulé.

TRANSIMS [68] est un simulateur microscopique fondé sur un automate cellulaire. Chaque véhicule est placé dans une cellule, représentant une portion de route. À chaque pas de temps, il avance d'un certain nombre de cellules, selon sa vitesse maximale. Comme une cellule ne peut être occupée que par un seul véhicule, si la cellule devant lui est occupée, il devra attendre qu'elle se libère.

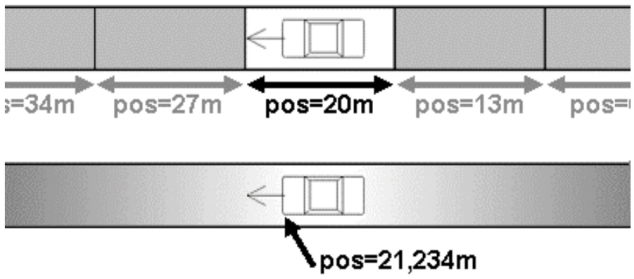

Figure 1.8: Automate cellulaire vs espace euclidien [47]

SUMO [10] est un simulateur multi-agent et multimodal. L'espace est géré cette fois-ci de manière continue (espace euclidien, voir fig 1.8). SUMO est un simulateur purement microscopique. Chaque véhicule est défini explicitement par un identifiant unique et a son propre itinéraire dans le réseau. Les véhicules calculent leurs vitesses en tenant compte de la position, de la vitesse et de l'accélération du véhicule qui le précède. Cette méthode est appelée un "modèle de poursuite".

ARCHISIM [18] utilise également un modèle de poursuite, mais il a la particularité de coupler la simulation de trafic à un simulateur de conduite. Ainsi, des utilisateurs humains vont être amenés à interagir avec les agents logiciels. Ceci permet d'introduire dans la simulation des comportements humains réalistes.

1.3.3 La simulation mésoscopique

Les modèles se situant entre le niveau de détail élevé de la simulation microscopique et faible de la simulation macroscopique sont appelés mésoscopiques. On classe dans cette catégorie les modèles reprenant des éléments venant du monde micro aussi bien que macro.

Ces modèles permettent la simulation de grands réseaux de transport tout en conservant les détails propre au niveau individuel. Par exemple, l'agrégation des véhicules en petits groupes considérés homogènes, permet de passer d'un modèle microscopique, ou chaque individu est simulé, à un modèle mésoscopique, où l'on considère désormais les convois de véhicules.

AIMSUN [6], depuis sa version 6, est un simulateur hybride. Il modélise les interactions inter-véhiculaires importantes avec un niveau de détail fin, en utilisant un modèle de

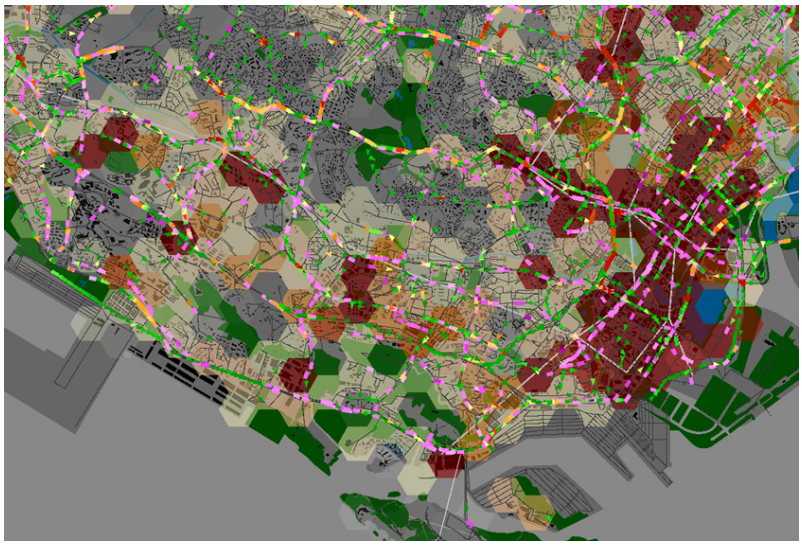

Figure 1.9: Un exemple de simulation mésoscopiques avec MATSim

poursuite et de changement de voie. Le reste du temps, il traite le trafic comme un flux, ce qui permet à l'utilisateur de modéliser les aspects dynamiques de très grands réseaux en faisant abstraction d'une grande partie des contraintes de calibration.

MATSim [63] a été conçu depuis le début dans le but de simuler de larges scénarios afin d'optimiser la demande des voyageurs. L'approche de MATSim est fondée sur un système de files d'attente, qui symbolise la capacité d'une portion de route. Si un agent veut entrer sur une portion qui a atteint sa capacité maximale, il devra attendre que de la place se libère. Un véhicule naviguera librement au maximum de sa vitesse quand il sera sur une portion de route. On peut considérer MATSim comme étant mésoscopique, car bien que chaque agent soit représenté individuellement, leurs déplacements dans le réseau ne dépendent pas de causes microscopiques.

SM4T [86], un simulateur de voyages multi-modal développé récemment, a un fonctionnement hybride similaire : chaque utilisateur y est représenté individuellement selon le paradigme multi-agent, mais le flux d'usagers est modélisé grâce au diagramme fondamental du trafic. Cette conception permet ici encore de représenter un nombre important d'usagers dans des réseaux de tailles conséquentes. Simuler chaque usager permet en outre de modéliser des comportements complexes où les décisions individuelles sont importantes, tels que le covoiturage ou le partage de places de parking.

1.4 Conclusion

La gestion du trafic routier est un problème complexe. Le simuler permet de concevoir et de tester l'efficacité de politiques de gestion sans risquer de perturber le fonctionnement

du réseau réel.

Nous avons vu dans ce chapitre que le paradigme multi-agent était particulièrement adapté pour la simulation de trafic. Il permet en effet de modéliser les comportements individuels de chacun des usagers, afin d'étudier leurs impacts sur l'état du réseau routier. Il existe à ce jour de nombreux simulateurs de trafic multi-agents. Il ne permettent cependant pas de modéliser avec un niveau de détail important des régions étendues, impliquant de nombreux usagers.

Chapitre 2

Le calcul haute performance

Sommaire

[!h]

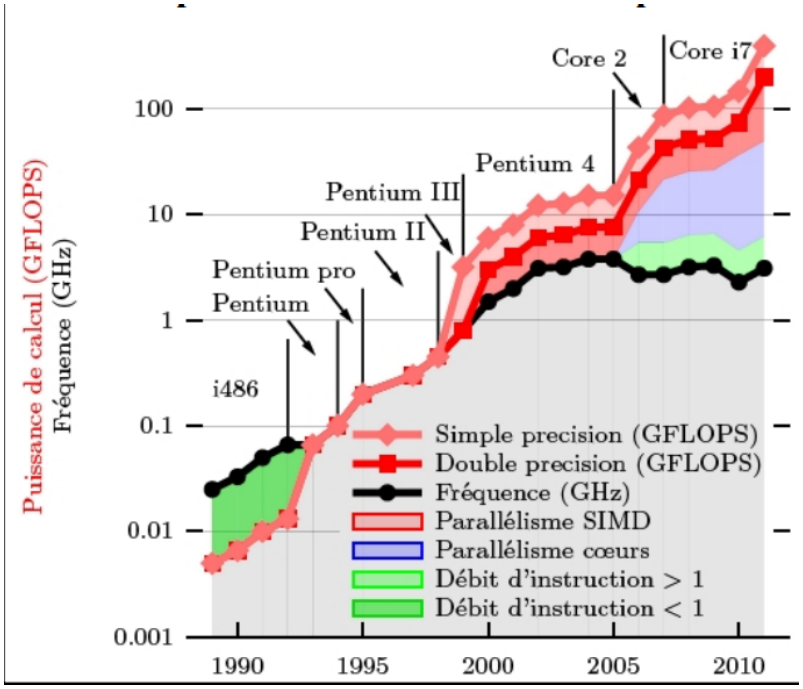

Figure 2.1: L'amélioration des performances des CPU est due à la parallélisation depuis 2004

2.1 Introduction

Gordon Moore, le co-fondateur d'Intel, émit en 1965 une célèbre conjecture empirique prophétisant que le nombre de transistors présents dans un processeur doublerait tous les ans. Plus tard, cette assertion fut étendue par David House qui affirma que la puissance de calcul elle même doublerait tous les dix-huit mois. Bien qu'au départ, ces affirmations n'avait pour but que la prévision pour les dix prochaines années, elles se sont révélées étonnamment exactes jusqu'en 2004. À cette date, la loi de Moore a cessée d'être vérifiée. Les limites physiques de la miniaturisation ont été atteintes, et il est devenu de plus en plus difficile d'améliorer les performances d'un microprocesseur.

Mais les ingénieurs ont trouvé des solutions pour pallier ce problème, en distribuant les calculs sur plusieurs ordinateurs individuels, ou encore sur des processeurs disposant de plusieurs cœurs de calcul. Le problème à traiter est divisé en plusieurs tâches plus simples, chacune attribuée à une UC (Unité de Calcul). Ces architectures ont permis de continuer à améliorer les performances des systèmes informatiques, et la loi de Moore se trouve jusqu'à aujourd'hui vérifiée (fig 2.1).

La puissance de calcul rendue disponible grâce aux architectures parallèles fut longtemps réservée à des domaines comme la météorologie ou le nucléaire. Les solutions utilisées

étaient en effet extrêmement coûteuses. Les centres de recherche et les industries devaient disposer de super-calculateurs à la pointe de la technologie, et devaient développer eux-même les logiciels permettant d'exploiter ce matériel. Mais aujourd'hui, les processeurs standards sont devenus très puissants, et il est possible de les combiner pour disposer de machines très performantes à coûts réduits. Certaines entreprises proposent même de louer leurs infrastructures (*cloud computing*), ce qui permet de disposer d'une puissance à la demande. De plus, le développement de logiciels libres permettant de facilement déployer des calculs sur ce type de matériel permettent encore de réduire les coûts de mise en œuvre de calculs haute-performance.

Le calcul haute-performance est devenu aujourd'hui un élément crucial du développement économique et technologique, et sa démocratisation a permis l'émergence de nombreux champs d'application. Il est utilisé pour étudier des phénomènes d'une grande complexité, telle que les activités économiques, les processus physiques et chimiques du vivant.

Le calcul parallèle désigne l'opération de faire coopérer plusieurs UC pour réaliser un calcul. Cette manière de procéder apporte plusieurs avantages. Elle permet tout d'abord d'augmenter la vitesse d'exécution du programme, avec dans l'idéal un temps de calcul divisé par le nombre d'UC à notre disposition (bien que cela ne soit jamais le cas, comme on le verra plus tard). Il permet également d'augmenter la taille de la mémoire que l'on a à sa disposition. En effet si l'on répartit un calcul sur N stations individuelles, on dispose d'autant de fois plus de mémoire.

La parallélisation d'un programme introduit de nouvelles problématiques. Pour que la parallélisation soit efficace, il faut gérer le partage des tâches. C'est à dire découper le problème en sous-tâches, et les répartir entre les différentes UC. Les tâches n'étant pas indépendantes (il ne s'agirait pas de programmation parallèle dans le cas contraire), il est donc nécessaire de mettre en place des routines permettant l'échange d'information entre les processus. La manière de procéder dépend grandement du type d'architecture dont on dispose.

2.2 Les différentes architectures parallèles

Michael Flynn a proposé une classification des différentes architectures parallèles existantes [34]. Il distingue quatre catégories d'architecture, classées selon leur flux d'instructions et de données :

- **SISD (Single Instruction, Single Data)** : Un ordinateur séquentiel, dans lequel un unique processeur exécute un unique flot d'instruction sur des données résidant dans une mémoire unique. Il s'agit de l'architecture de Von Neumann.
- **SIMD (Single Instruction, Multiple Data)** : La même instruction est appliquée

[!h]

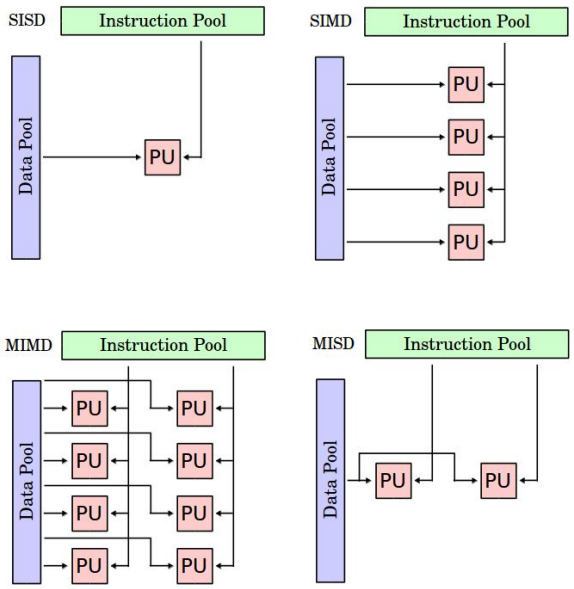

Figure 2.2: Taxonomie de Flynn (Wikipédia)

à plusieurs données à la fois. Particulièrement utilisé pour le calcul matriciel.

- **MIMD (Multiple Instruction, Multiple Data)** : Plusieurs processeurs effectuent des calculs sur des données différentes, placées sur des mémoires distinctes.
- **MISD (Multiple Instruction, Single Data)** : Une donnée unique est traitée simultanément par plusieurs processeurs en parallèle. Cette catégorie se retrouve rarement dans la pratique.

La figure 2.2 illustre ces quatre architectures. Les deux architectures utilisées pour l'exécution de tâches dans un environnement de calcul haute-performance sont les modèles SIMD et MIMD.

L'architecture MIMD est traditionnellement la plus utilisée et la plus étudiée. Il existe deux grandes catégories de modèles tombant sous la classification MIMD : les modèles à mémoire partagée, où tous les processeurs ont accès à l'ensemble de la mémoire, et les modèles à mémoire distribuée, où chaque processeur possède sa propre mémoire, et n'a pas accès à celles des autres.

L'architecture SIMD est quant à elle de plus en plus utilisée, principalement grâce aux récents progrès effectués dans le domaine de la programmation sur carte graphique. Des interfaces de programmation ont en effet été développées, qui permettent d'exploiter leur

[!h]

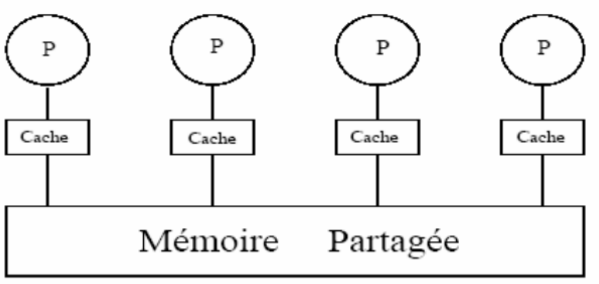

Figure 2.3: Architecture à mémoire partagée

puissance pour le calcul haute performance. La programmation sur processeurs graphiques s'est révélée particulièrement efficace pour certains problèmes.

2.3 Architecture à mémoire partagée

Une machine MIMD à mémoire partagée est composée de multiples processeurs évoluant de façon asynchrone. Il vont communiquer par le biais de lectures/écritures dans une mémoire unique qu'ils partagent (figure 2.3). La synchronisation des exécutions des unités de calculs est souvent nécessaire, pour assurer la cohérence des données.

2.3.1 Les processus légers

Les premières implémentations de ce type étaient assurées grâce à l'utilisation de processus légers. La librairie de multithreading POSIX Pthreads, adoptée par la plupart des systèmes d'exploitation, implémente la notion de *threads*. Ces processus légers peuvent être distribués sur plusieurs unités de calculs pour une exécution simultanée. Il sont cependant difficile à gérer, et l'utilisation de *threads* n'assure pas une exécution parallèle. L'utilisation de *threads* pour la parallélisation d'un programme implique un nombre considérable de lignes de code spécifiquement dédiées. Il est nécessaire de déclarer les structures qui lui sont propres, de créer les *threads* eux mêmes, de calculer les bornes des boucles, etc.

Il y eu différentes tentatives d'automatiser la parallélisation de code grâce aux compilateurs (par exemple *High Performance Fortran*). Cependant, la conversion de programmes séquentiels en programmes parallèles est une tâche très complexe, et la plupart des compilateurs ont démontré des performances limitées. De plus, chaque vendeur proposait son propre ensemble de directives, et les programmes n'étaient ainsi pas portables.

2.3.2 OpenMP

Plusieurs tentatives de standardisation ont été proposées mais n'ont jamais été adoptées par les instances de normalisation. En 1997, les industriels et les constructeurs ont adopté comme standard une implémentation du multithreading nommée OpenMP. OpenMP est un ensemble de directives de compilation explicite pour paralléliser un programme, qui possède une interface dans plusieurs langages. OpenMP propose ainsi une interface de haut niveau pour une programmation parallèle de type SIMD (Single Program Multiple Data) sur machine à mémoire partagée. Il s'est imposé aujourd'hui comme l'un des deux grands standards du calcul scientifique.

La forme la plus simple de parallélisme offerte par OpenMP est la déclaration de portions de code parallèles (généralement les boucles du programme séquentiel). Ces régions sont introduites en insérant explicitement dans le code séquentiel des directives de compilation propres à OpenMP. En C++, par exemple, les zones parallèles sont déclarées grâce la directive préprocesseur pragma, comme dans le code suivant :

```
#include <omp.h>

int main()
{
    const int size = 256;
    double sinTable[size];
#pragma omp parallel for
    for(int n=0; n<256; ++n)
        sinTable[n] = std::sin(2 * M_PI * n / size);
}
```

À l'entrée de chaque région parallèle, le *thread* maître crée des *threads* esclaves selon les besoins, exécutant chacun une tâche implicite de manière concurrente. Chaque *thread* exécute sa propre séquence d'instructions et c'est le système d'exploitation qui détermine l'ordre d'exécution des *threads* et les affecte aux unités de calcul disponibles. Les *threads* Esclaves sont finalement joints à la sortie de la zone parallèle (figure 2.4). Le programme séquentiel est ainsi modifié incrémentalement, jusqu'à ce que toutes ses boucles soient parallélisées.

La mémoire est commune à l'ensemble des processus. Les variables globales sont ainsi simplement partagées par les *threads*. Cependant, tout n'est pas partagé, et chaque processus dispose d'une mémoire locale. Il est ainsi important de déterminer si une donnée doit être privée ou partagée avec les autres processus. La déclaration de la visibilité d'une variable est faite lorsqu'une région parallèle est déclarée. Une variable privée ne peut être vue que par un unique *thread*, et reste indéfinie pour la région parallèle. Par exemple, les

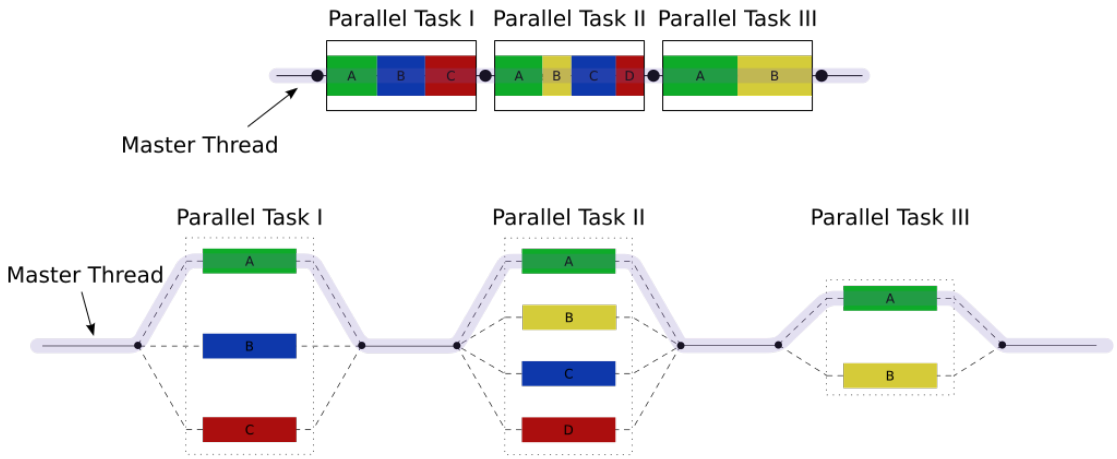

Figure 2.4: Gestion des *threads* avec openmp (Wikipédia)

variables de la pile des fonctions appelées à l'intérieur d'une section parallèle seront toujours privées, ainsi que les variables temporaires, ou les compteurs de boucle. Les variables partagées, quant à elles, peuvent être vues par l'intégralité des *threads*. Les communications entre les *threads* se font via ces variables partagées. Il faut toutefois être prudent, car des problèmes de synchronisation peuvent corrompre l'intégrité des données, si par exemple plusieurs *threads* tentent simultanément de modifier une donnée. C'est pourquoi des directives de synchronisation doivent parfois être explicitées par le programmeur.

2.3.3 Conclusion

Les programmes multithreads développés avec Pthread dépendent de primitives spécifiques à la plateforme, ce qui n'est pas le cas d'OpenMP, qui est très portable. Il permet en outre de simplifier la mise en œuvre de code parallèle pour le programmeur. Les données sont en effet gérées automatiquement par les directives, et la conversion de code séquentiel ne nécessite pas de changements drastiques, ce qui réduit les chances d'introduire des bugs. Ce standard offre donc de gros avantages de performance et de simplicité, et est la solution à privilégier lors de la programmation de système MIMD à mémoire partagée.

Cependant OpenMP ne permet pas de gérer finement la distribution des *threads* sur les différents cœurs de calcul. Il faut en outre gérer efficacement les synchronisations inhérentes à tout programme parallèle. Certains bugs liés à des problèmes de synchronisation peuvent être difficiles à détecter et à résoudre. De plus, les systèmes à mémoire partagée sont dépendants d'une architecture spécifique, souvent coûteuse, et ne peuvent par exemple pas être déployés sur un cluster de calcul.

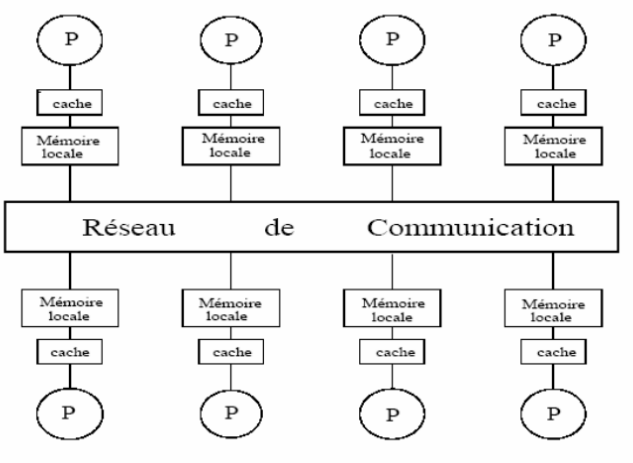

Figure 2.5: Architecture à mémoire distribuée

2.4 Architecture à mémoire distribuée

En réalité, à part les processeurs multicœurs, peu de systèmes informatiques offrent véritablement une mémoire partagée. La plupart des supercalculateurs utilisent une mémoire répartie en plusieurs nœuds, disposant chacun d'un processeur la gérant. La mémoire d'un tel système est dite distribuée : chaque portion n'est accessible qu'à certains processeurs. Les clusters de calcul sont l'exemple le plus fréquent des architectures à mémoire distribuée.

Un réseau de communication relie les différents nœuds, qui doivent communiquer entre eux par des échanges de messages de manière non instantanée. Ce modèle de programmation est appelé envoi de message (*message passing*). Chaque nœud de calcul, composé d'un processeur et d'un bloc mémoire, va exécuter un programme séquentiel utilisant des procédures d'envoi et de réception de messages. Ces envois de messages sont le seul méchanisme de synchronisation disponible pour le système.

Bien sûr, ces communications engendrent un surcoût, qui dépend du réseau de communications et de l'implémentation utilisée. Dans certains cas, les communications prennent en effet tellement de temps par rapport aux temps de calcul que la parallélisation s'avère contre-productive (le programme séquentiel est plus rapide que sa version parallélisée). Il est possible de contrebalancer ces surcoûts en investissant dans de meilleurs infrastructures réseaux, mais d'une part cela se révèle onéreux, et d'autre part il existera toujours une limite technologique à l'efficacité des équipements réseaux. Pour que la parallélisation soit efficiente, il convient donc de minimiser la quantité des communications.

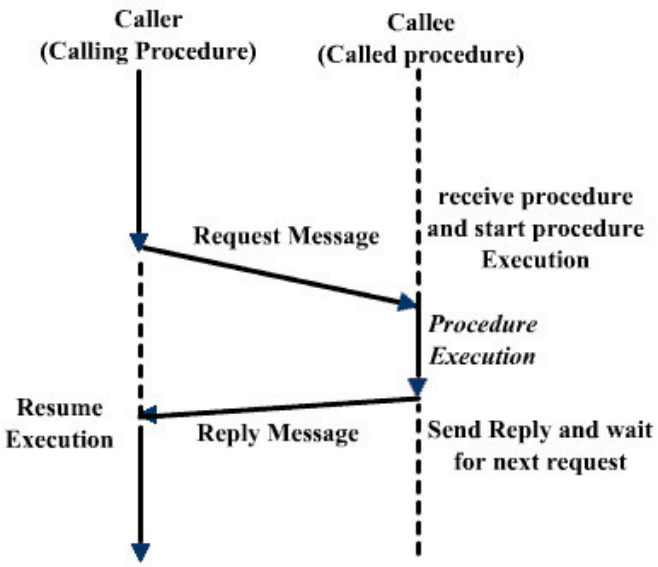

Figure 2.6: Modèle RPC

2.4.1 RPC

L'un des premiers modèles permettant l'exécution de code de manière distribuée fonctionnait selon un principe client-serveur. Un nœud maître (le client), disposant de l'intégralité du code, délègue une partie de son travail à des nœuds esclaves distants (les serveurs), en leur fournissant des procédures à exécuter. Les esclaves font alors leurs calculs et retournent le résultat au maître, qui peut alors continuer le déroulement de son programme (figure 2.6).

Un tel fonctionnement est appelé RPC (*Remote Procedure Call*). Les bases théoriques en fûrent mises en place dès les années 70, mais il a fallu attendre les années 80 pour voir naître les premières implémentations. RPC permet d'utiliser un modèle de programmation classique basé sur l'appel de procédures. Ses implémentations fournissent au programmeur une interface qui masque la complexité du réseau et de l'envoi/réception de messages. Il s'agit d'une abstraction de haut niveau, qui facilite grandement la programmation de systèmes distribués. Cependant le modèle RPC n'est pas adapté à tous les types de problèmes, et il a tendance à générer beaucoup de communications et de temps d'attente entre les processus.

2.4.2 MPI

De nombreuses librairies permettant de gérer plus finement les communications que ne le fait RPC ont vu le jour. Grâce à elles, le développeur peut explicitement contrôler les messages qui seront échangés entre les cœurs de calcul. La plupart de ces librairies

proposaient des modèles de passage de messages très similaires, avec seulement quelques différences mineures entres elles. C'est pourquoi un forum de chercheurs et d'industriels s'est réuni pour définir une interface standard de passage de messages, afin d'assurer la portabilité des applications parallèles sur des machines à mémoire distribuée. C'est ainsi, grâce à la collaboration d'une quarantaine d'organisations que naquit MPI (*Message Passing Interface*) en 1994.

MPI est un ensemble de normes qui furent implémentées dans de nombreux langages. En plus de faciliter la programmation par envoi de messages, un programme MPI peut être déployé de manière transparente sur une architecture hétérogène. MPI demeure encore aujourd'hui le standard *de facto* de la programmation sur systèmes distribués.

Pour déployer un calcul sur un système à mémoire distribuée comportant N nœuds numérotés de 0 à N-1 (le rang du nœud). On découpe le domaine global de résolution en N sous-domaines de tailles la plus homogène possible. Un processus est alors affecté bijectivement à un sous-domaine et à un cœur de calcul. Les processus exécutent simultanément le même programme en parallèle et s'échangent des données aux interfaces des sous-domaines.

Le programme est écrit dans un langage classique (Fortran, C, C++, etc.). L'intégralité des variables du programme sont privées et sont stockées dans la mémoire locale propre à chaque processus. Bien que le programme soit le même pour chaque processus, chacun d'eux en exécute un partie différente, selon son rang. Voici un exemple de code MPI, écrit en C.

```
#include <mpi.h>
main(int argc, char **argv)
{
    int my_id;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    if( my_id == 0 ) {
        /* do some work as process 0 */
    }
    else if( my_id == 1 ) {
        /* do some work as process 1 */
    }
    else {
        /* do this work in any remaining processes */
    }
    MPI_Finalize();
}
```

Les communications se font à l'aide d'envoi et de réception de messages entre les processus. Un processus peut envoyer un message à un autre en fournissant le rang du processus et un identifiant pour le message. Le processus receveur déclare ensuite recevoir le message et traite les données reçues. Ce mode de communication impliquant un envoyeur et un receveur unique est appelé communication point à point.

Il y a de nombreux cas où un processus a besoin de communiquer avec plusieurs autres, voire avec l'ensemble des processus. C'est le cas par exemple du modèle maître-esclave. Dans ce cas, il serait contraignant d'écrire explicitement l'intégralité des envois/réceptions. Le réseau serait de plus utilisé de manière sous-optimale. C'est pourquoi MPI fournit également un ensemble de procédures permettant ce type de communications collectives.

2.4.3 Le cloud computing

Cette dernière décennie a vu émerger un nouveau paradigme de systèmes informatiques, appelé *cloud computing*. Des serveurs reliés par une infrastructure de communication sont loués à la demande selon les besoins de l'utilisateur. Là où les calculs gourmands étaient autrefois effectués au sein même de l'entreprise, dans un cluster qu'elle avait mis en place elle-même, ils sont désormais effectués avec une grande souplesse sur des équipements externes.

Un ensemble de serveurs, virtualisés ou non, une grande capacité de stockage et une infrastructure réseau de grande qualité sont ainsi disponibles à tout moment et à un coût réduit pour les utilisateurs. Le cloud computing facilite ainsi l'accès pour un large public à une machine distribuée haute performance.

2.4.4 Conclusion

Certains problèmes demandent une grande puissance de calcul et une quantité importante de mémoire pour être résolus. Les architectures à mémoire distribuée offre une solution performante pour leur résolution. Elles permettent en effet d'utiliser simultanément un nombre virtuellement illimité de stations informatiques bon marché reliées par un réseau. Cela est d'autant plus vrai avec l'émergence récente du cloud computing, qui permet de disposer rapidement et à faible coût d'une infrastructure haute performance.

MPI fournit une manière puissante, efficace et portable d'exprimer des programmes parallèles sur des systèmes à mémoire distribuée. Le programmeur doit gérer en détail la manière dont les processus communiquent. Bien que cet aspect peut être difficile, cela permet un grande flexibilité. Il est depuis 15 ans, le standard *de facto* utilisé sur ce type d'architectures, tout comme l'est OpenMP sur les systèmes à mémoire partagée.

2.5 Le GPGPU

Les cartes graphiques (*Graphics Processing Units*, ou GPU en anglais) ont connu un essor important ces dernières décennies, poussé par l'industrie du jeux vidéo. Elles étaient à l'origine exclusivement utilisées pour l'affichage d'images de synthèse. Une carte graphique est un système SIMD. Il dispose d'une unité global de contrôle et de nombreux processeurs. Chacun de ces processeurs exécute le même programme simultanément, mais ce programme est appliqué à des données différentes. Ceci est particulièrement utile dans les situations où de nombreux calculs similaires sont nécessaires. Les GPU offrent ainsi une grande puissance de calcul pour un faible coût, ce qui leur a valu d'être de plus en plus utilisés pour le calcul scientifique. Cet usage détourné est appelé GPGPU, pour *General Purpose* GPU.

Les GPU sont à l'origine destinés au traitement d'images, et sont optimisés pour traiter vertex et pixels. Les limitations matérielles propres à ce type de matériel impose au programmeur de se restreindre à des structures de données et des primitives très simples. Le portage d'algorithmes classiques sur GPU nécessite donc de fournir un effort de développement important. Ceci est plus particulièrement vrai si l'on utilise les bibliothèques spécifiques au développement graphique, telles OpenGL ou DirectX.

2.5.1 CUDA

En raison des difficultés de portage d'algorithmes traditionnels sur GPU, et du manque d'interopérabilité des programmes résultants, le GPGPU ne fut pas très développé jusqu'en 2007. C'est à cette date que NVIDIA introduisit CUDA (*Compute Unified Device Architecture*), la première bibliothèque fournissant une interface de programmation pour carte graphique de haut niveau.

Il s'agit d'une extension du langage C, qui offre une courbe d'apprentissage beaucoup plus élevée qu'avec les technologies disponibles jusqu'alors. CUDA permet d'accéder à certaines caractéristiques matérielle qu'il était impossible d'utiliser avant, et un code écrit en CUDA est plus propre et facile à maintenir. Le programme est écrit en version séquentielle dans le langage C. On ajoute ensuite des sous-routines spécifiques qui seront exécutées sur le GPU. La partie séquentielle du programme lance ces sous-routines et détermine le nombre de *threads* qui seront exécutés.

2.5.2 OpenCL

Le succès de CUDA a ouvert la voie à OpenCL (*Open Computing Language*), une API standard et libre. OpenCL fournit une abstraction de plus haut niveau que CUDA. Il permet de gérer des ressources hétérogènes : un GPU, un CPU, ou un processeur multi-

cœurs sont tous considérés comme des vecteurs potentiels du calcul parallèle.

Ainsi, pour une classe étendue de problèmes, un code OpenCL bien écrit peut être porté facilement sur des architectures très différentes. Cela permet également d'utiliser simultanément les différentes ressources d'un ordinateur. OpenCL est largement supporté par l'industrie, et n'est donc pas, à l'inverse de CUDA, conditionné par le succès ou l'échec d'une seule et même organisation.

2.5.3 Conclusion

L'architecture GPU est optimisée pour des applications massivement parallèles, s'exécutant sur des milliers de *threads*. Cependant, les processeurs contenus dans un GPU ont une fréquence assez faible, et ne dispose pas de certaines fonctionnalités que l'on trouve sur un CPU. Donc, bien que le nombre de *threads* s'exécutant sur un GPU soit très élevé, chacun d'eux à une performance d'exécution assez faible.

Le modèle de programmation sur GPU est ainsi plus adapté a certains types d'applications, comportant des données hautement parallèles. Il se révèle pour ce type d'application plus efficace que la programmation sur CPU multi-cœurs. Certains travaux [50, 64, 79] ont su tirer parti avec succès de l'architecture GPU pour améliorer les performances de simulations multi-agents. Cependant, la simulation sur GPU exige une structure de donnée très particulière. Pour la majorité des problèmes, la programmation sur CPU reste plus flexible.

2.6 Limitations du calcul parallèle

La parallélisation offre une puissance de calcul nouvelle, qui permet d'augmenter considérablement l'efficacité des systèmes. Cette puissance de calcul n'augmente toutefois pas de manière linéaire avec la puissance du matériel dont on dispose. Le passage à l'échelle (scalabilité) d'un programme parallèle désigne l'augmentation des performances obtenue lorsque l'on ajoute des cœurs de calcul. La scalabilité est intrinsèquement liée à la capacité du problème à être parallélisé.

Soit T_1 le temps d'exécution séquentiel du programme, P le nombre de processus parallèles, et T_P le temps d'exécution en parallèle. L'accélération S_P exprime combien de fois le programme parallèle est plus rapide que sa version séquentielle : $S_P = \frac{T_1}{T_P}$. L'efficacité E_P mesure l'accélération par rapport au nombre de processus : $E_P = \frac{S_P}{P}$; L'optimum théorique pour ces deux métriques est de $S_P = P$ et $E_P = 1$. C'est ce qui se produit dans le meilleurs des cas, lorsque le programme est d'autant de fois plus rapide que l'on dispose d'unités de calcul. Malheureusement, dans la grande majorité des cas, $E_P < 1$. Ceci est dû à plusieurs raisons :

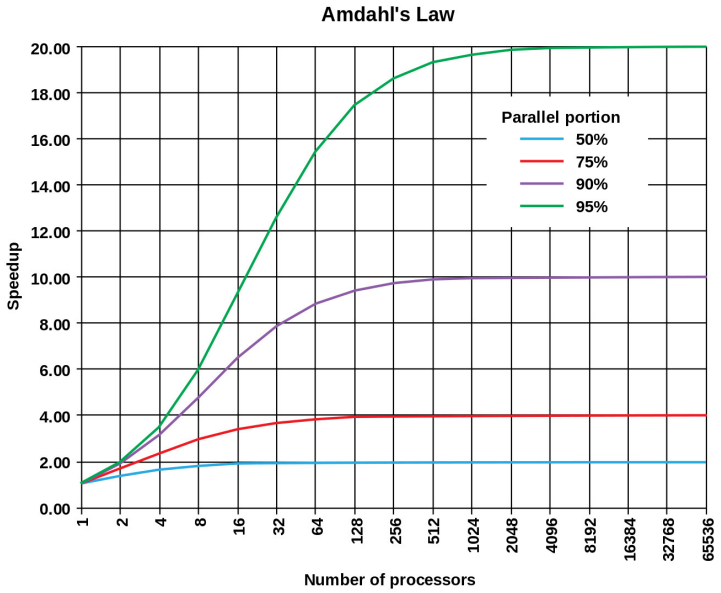

Figure 2.7: Loi d'Amdahl (Wikipédia)

- Temps de communication entre processus
- Temps d'attente entre les processus lorsque les tâches ne sont pas équilibrées,
- Section du programme non parallélisable, exécutée en séquentiel sur un seul processus,

Comme on l'a vu précédemment, lorsque l'on parallélise un programme, l'échange d'information entre les différents processus est essentiel. Pour optimiser l'efficacité de la parallélisation, les temps de communication doivent aussi être réduits; en général, un temps de communication est la somme d'un temps fixe dit de latence et d'un temps de transfert proportionnel à la taille du message. Il faut donc réduire à la fois le nombre et le volume des communications.

D'autre part, la charge de travail de chacun des processus n'est généralement pas exactement la même, et ils doivent ainsi s'attendre les uns les autres en différents points de l'exécution, pour se synchroniser et assurer la consistance du calcul. Pour éviter ces temps d'attente, il est important de répartir la charge de travail de manière équilibrée tout au long de l'exécution du programme.

Par ailleurs, la grande majorité des programmes parallèles dispose d'une partie de son code qu'il est nécessaire d'exécuter de manière séquentielle. C'est Gene Amdhal qui pour la première fois énonça précisément les relations existantes entre l'efficacité d'un programme parallèle et la quantité de code séquentiel qu'il contient [3].

On considère que l'on se situe dans un système idéal où les temps d'attente et de communication sont nuls. On souhaite obtenir une estimation de la borne supérieure du temps d'exécution. Soit a la proportion du programme exécutée en parallèle.

On a :

$$T_P = a\frac{T_1}{P} + (1-a)T_1 \tag{2.1}$$

Soit, avec $S_P = \frac{T_1}{T_P}$:

$$S_P = \frac{P}{(1-a)P + a} \tag{2.2}$$

On a donc :

$$S_P = \begin{cases} P & \text{si} \quad a = 1 \\ 1 & \text{si} \quad a = 0 \end{cases} \tag{2.3}$$

Pour a fixé, l'accélération est inférieure à la valeur asymptotique :

$$\lim_{P \to +\infty} S_P = \frac{1}{1-a} \tag{2.4}$$

Ainsi, par exemple, pour $a = 0.5$, on a $S_P \leq 2$ et pour $a = 0.9$, on a $S_P \leq 10$. La figure 2.7 montre les limites atteintes pour le gain de performance en fonction de la quantité de code séquentiel présent dans le programme.

On constate que pour que la version distribuée soit efficace, c'est à dire que les unités de calcul à notre disposition soit correctement exploitées, il est nécessaire de paralléliser une part très importante de l'algorithme. L'enseignement principal de la loi d'Amdahl est que la performance d'un algorithme parallèle est limitée fortement par la proportion du programme à s'exécuter en séquentiel. Même si l'on dispose d'une quantité de ressources très importante, si le programme n'est pas hautement parallélisable, les résultats seront décevants.

2.7 Le calcul haute performance au service de la simulation

Comme nous l'avons vu, les chercheurs utilisent la simulation pour résoudre des problèmes qu'il serait difficile, voire impossible de résoudre avec les méthodes analytiques traditionnelles. Toutefois, la simulation de systèmes complexes de taille importante et impliquant un niveau de détail élevé prend beaucoup de temps à s'exécuter sur une architecture séquentielle.

C'est pourquoi le calcul parallèle joue un grand rôle dans le présent et le futur de la simulation sur ordinateur. Avec ces récents développement, tant au niveau matériel (GPGPU, cloud computing, processeurs multi-cœurs), qu'au niveau logiciel (standards de programmation), il permet de réduire considérablement le temps nécessaire à l'obtention

des résultats. Il permet ainsi de simuler des systèmes hautement complexes en temps réel, ce qui n'était pas envisageable sur un simple CPU.

2.7.1 Les plateformes multi-agents parallèles

Comme nous l'avons vu dans le chapitre précédent, le paradigme multi-agent est intrinsèquement distribué. La nature même du paradigme agent implique que la portion de code parallélisable est proche des 100%. C'est donc tout naturellement que les SMA sont considérés comme un technologie prometteuse pour faire face aux défis posés par la distribution à large échelle de systèmes complexes, tels que la simulation de systèmes financiers, la prévision météo, la modélisation de bactéries, ou encore la simulation de trafic routier et aérien.

La question qui se pose dans ce contexte est la suivante : comment les SMA peuvent ils être intégrés dans des environnements de calcul haute performance ? Il s'agit en effet d'une tâche complexe, due aux particularités techniques propres aux SMA. Dans de tels systèmes, les agents individuels communiquent de manière intensive entre eux. Il modifient en outre leur environnement, ce qui peut avoir une conséquence dans la prise de décision future des autres agents. Il est donc nécessaire, en termes de parallélisation, de partager de nombreuses données entre les processus.

C'est pourquoi les chercheurs ont développé des solutions permettant de gérer la complexité inhérente à la collecte d'informations et à la synchronisation entre les processus dans la distribution de SMA. Il existe ainsi de nombreuses plateformes facilitant le développement de simulations multi-agent s'exécutant sur des architectures parallèles. Nous décrirons ici quelques unes des ces plateformes les plus utilisées. Leurs caractéristiques princpales sont résumées dans le tableau 2.1.

	RepastHPC	D-Mason	FLAME	Pandora	Jade
Lang. Prog	C++	Java	XMML/C	C/C++	Java
Type de simulation	event-driven	time-driven	time-driven	time-driven	time-driven
Communication	MPI	JMS	MPI	MPI	RMI

Tableau 2.1: Comparaison des différentes plateformes multi-agents distribuées [48]

2.7.1.1 RepastHPC

RepastHPC [20] a été développé par le laboratoire d'Argonne. Il a été conçu pour un environnement haute-performance, dans lequel les agents sont répartis sur de nombreux processeurs. Il est fondé sur la plateforme Repast symphony, et adapte son système de projection à un environnement distribué. Tout comme Repast, il est extrêmement flexible, et permet même de changer dynamiquement le modèle utilisé pendant l'exécution.

Repast HPC est écrit en C++, et, la mémoire n'étant pas partagée, il utilise MPI pour la communication inter-processus. Toutes les synchronisations et les communications sont gérées automatiquement par RepastHPC. Ceci rend le développement de simulations distribuées bien plus facile, mais empêche la gestion fine des communications par le développeur.

2.7.1.2 D-Mason

D-Mason[21] (Distributed Mason) est développé par l'université de Salerne. Il s'agit, comme son nom l'indique, d'une version distribuée de la plateforme de simulation multi-agent MASON, qui permet de paralléliser le code de MASON déjà existant en apportant seulement quelques modifications mineures. MASON est un simulateur à évènements discrets. La temporalité des évènements est donc cruciale au bon déroulement de la simulation. D-Mason fonctionne selon une approche optimiste, relâchant les contraintes temporelles. Il permet aux évènements d'être traités dans le mauvais ordre, rectifiant ensuite si nécessaire.

D-Mason adopte une stratégie de découpage de l'espace pour répartir le travail entre les unités de calcul, afin de limiter les communications nécessaires à la synchronisation. Le découpage est défini explicitement par l'utilisateur. Comme les agents peuvent se déplacer d'une partition à une autre, D-Mason fournit un mécanisme d'équilibrage de charge pour éviter que tous les agents ne se retrouve sur la même partition.

Il fonctionne selon le principe maître-esclave. Un nœud maître assigne une portion des calculs à chacun des esclaves. Pour chaque pas de simulation, les esclaves simulent les agents qui leur sont assignés et envoient les résultats aux autres esclaves concernés. Ils utilisent pour cela JMS ActiveMQ, une librairie de passage de messages de Java. Cette solution n'est pas la plus extensible des solutions dans un environnement HPC. C'est pourquoi une version décentralisée de D-Mason [22] a plus tard été proposée, utilisant cette fois MPI, permettant un meilleur passage à l'échelle.

2.7.1.3 FLAME

FLAME [19], un framework multi-agent conçu dès le début pour les simulations parallèles a été développé par l'Université de Sheffield. Il a été conçu pour permettre à l'utilisateur de développer une large gamme de modèles d'agents sans avoir une quelconque expertise en programmation parallèle. Il fournit des spécifications sous la forme d'un cadre formel qui peut être utilisé par les développeurs pour créer des modèles et des outils. FLAME peut également s'exécuter sur GPU.

Les agents sont modélisés sous la forme d'automates à états finis. Ces automates peuvent recevoir et envoyer des messages (en utilisant MPI) à l'entrée et à la sortie de chaque état.

Bien que FLAME libère le programmeur de la gestion des communications, il lui incombe tout de même de définir correctement son modèle pour qu'il soit parallélisable.

2.7.1.4 Pandora

Pandora [4] est développé par le groupe de recherche de simulation sociale du Centre Supercomputing Barcelone. Il est utilisé pour la simulation d'anciennes sociétés et leurs relations avec les transformations environnementales. Il a été explicitement programmé pour permettre ce type de simulations, nécessitant l'exécution simultanée d'un grand nombre d'agents.

Il permet des simulations d'agents placés dans un espace 2D, en implémentant un système perfectionné d'information géographique. Cet espace est découpé et réparti équitablement entre les nœuds de calcul. Pandora utilise C++ pour définir le modèle de simulation. Le code MPI nécessaire aux communications est généré automatiquement. Tout comme RepastHPC, Pandora permet une grande marge de passage à l'échelle.

2.7.1.5 JADE

Pour finir, JADE [11], est un framework conçu par le laboratoire Télécom Italia. Il permet de développer des applications orientées agent entièrement compatibles avec la spécification FIPA (un standard de communication multi-agent visant l'interopérabilité des systèmes).

La simulation est implémentée en JAVA. Chaque agent est défini comme un *thread* JAVA et les agents peuvent être distribués sur plusieurs hôtes, qui peuvent être hétérogènes. Le seul prérequis est que chaque nœud fasse tourner une machine virtuelle JAVA. Les communications entre les nœuds s'effectuent une nouvelle fois grâce à MPI. Les communications entre les agents présents sur un même nœud sont gérés de manière efficace par JAVA.

2.7.2 La simulation de trafic distribuée

Comme pour tout calcul distribué, le principal problème pour la parallélisation de simulation de trafic est la synchronisation efficace des différentes parties de la simulation s'exécutant sur des machines différentes. L'application du paradigme multi-agent à la simulation d'un réseau routier à large échelle, impliquant des centaines de milliers d'usagers, impose des besoins spécifiques. Une simulation de trafic est un ensemble d'agents situés (ils ont une position déterminée dans leur environnement) et mobiles (ils peuvent s'y déplacer) évoluant sur un réseau routier. Il est indispensable de prendre en compte les spécificités de la topologie du problème lors de sa distribution. C'est pourquoi, au delà des plateformes de simulation multi-agents distribuées générales, développées pour permettre

un large champs d'applications, des solutions destinées spécialement à la simulation de trafic routier ont été investiguées.

Nagel et Rickert [67] ont proposé une version parallèle de TRANSIMS, implémentant un modèle de communication maître-esclave. Le réseau routier est découpé et réparti au début de la simulation entre les UC. L'architecture maître-esclave peut poser problème pour le passage à l'échelle.

D-SUMO (*Distributed* SUMO) fait tourner plusieurs instances de SUMO sur différents nœuds de calcul et gère les communications nécessaires (principalement la migration d'agents d'un nœud à un autre). L'environnement est découpé grâce à SPARTSIM [81], un algorithme hybride permettant le découpage de l'espace en prenant en compte la topologie du réseau se trouvant sur cet espace. D-SUMO se passe complètement d'entité centrale gérant les communications. Cette nature décentralisée lui assure une grande capacité de passage à l'échelle. On peut également citer [2], qui est un travail similaire et indépendant, visant à distribuer SUMO grâce au framework de partition de graphe METIS. Ce projet est cependant encore au stade préliminaire.

2.8 Conclusion

Nous avons vu dans le chapitre 1 que les simulateurs de trafic multi-agents traditionnels étaient incapables de simuler un très grand nombre d'usagers. La simulation de trafic peut aujourd'hui se reposer sur les immenses efforts fournis pour le développement du calcul parallèle. Cependant, bien que les solutions traditionnelles de calcul parallèle fassent gagner beaucoup de temps pour le développement de telles simulations, il reste quand même nécessaire de les adapter aux spécificités du trafic routier pour une efficacité optimale.

Des plateformes multi-agents distribuées ont récemment vu le jour. Elles sont cependant généralistes et n'offrent souvent que peu de contrôle à l'utilisateur quand à la manière dont les calculs sont répartis. Dans une simulation de trafic, les agents sont situés et mobiles au sein de leur environnement. Si cet environnement est distribué entre les différents nœuds de calcul, les agents vont devoir migrer d'un nœud à un autre au grès de leurs déplacements. La charge de travail de chacun des nœuds va donc évoluer dynamiquement. La migration d'agents peut créer un important déséquilibre de charge.

Quelques solutions spécifiques à la simulation de trafic parallèle ont été proposées. Dans chacune d'elles, l'environnement dans lequel évoluent les agents est découpé statiquement au début de la simulation. TRANSIMS implémente un système de répartition de charge dans le cas où le même scénario est répété un grand nombre de fois. À chaque itération, la charge s'adapte et est répartie différemment en fonction des itérations précédentes. Les différentes versions distribuées de SUMO répartissent la charge statiquement au début de la simulation mais n'embarquent pas de mécanisme dynamique.

L'aspect de la répartition dynamique de charge propre à la simulation de trafic n'est pour l'instant que peu traité dans la littérature. Il s'agira d'un des sujets d'étude principaux de cette thèse.

Partie II

Contributions

Chapitre 3

Simulateur de déplacements

Sommaire

3.1 Introduction

L'objectif de cette thèse est l'étude de la scalabilité des différents modèles de simulateurs de trafic existants. Nous avons vu au chapitre 1 que ces simulateurs se répartissaient en deux grandes catégories : les simulateurs microscopiques et macroscopiques. Nous ne voulons pas proposer de solutions qui seraient spécifiques à un simulateur en particulier. Pour autant, l'implémentation et le test de nos propositions sur l'intégralité des simulateurs existants ne sont pas envisageables en termes de temps de développement.

C'est pourquoi nous avons développé un simulateur de trafic de référence. Nous l'avons voulu très simple : notre but est qu'il puisse incorporer les éléments qui sont la substance de tout simulateur de trafic, tout en s'abstrayant de la gestion fine de détails uniquement

nécessaires pour la simulation dans un contexte opérationnel. La raison d'être de notre simulateur est uniquement d'être suffisamment représentatif pour permettre le test de méthodes de parallélisation; il n'a aucune prétention au réalisme parfait de la simulation.

Notre simulateur doit être capable de décrire des situations macroscopiques et microscopiques. Nous présentons ainsi un cadre de travail permettant de décrire les éléments qui sont communs aux deux paradigmes, c'est à dire la gestion générale de l'environnement et des agents; puis nous nous intéressons en détail aux particularités qui sont propres à chacun des paradigmes.

Une simulation multi-agent peut être vue comme un ensemble d'entités qui interagissent ensemble et avec un environnement, qui peut lui même être un ensemble d'agents. Dans le cas de la simulation de trafic, les agents évoluent dans un réseau routier. Ils vont calculer leur itinéraire pour se rendre de leur lieux de départ vers leur destination. Un agent évolue plus lentement dans une partie du réseau encombrée que sur une route déserte. Des accidents peuvent également survenir, et bloquer une parcelle de route. Ces aléas du réseau sont à prendre en considération.

3.2 Organisation générale du simulateur

Notre simulateur est décomposé en trois éléments principaux (figure 3.1). Le premier sert à configurer les différents paramètres de la simulation. Ces paramètres comprennent le nombre d'agents présents ainsi que la durée de la simulation. Il charge le fichier contenant les données relatives au réseau de transport, crée les agents et les positionne sur le réseau. Il lance ensuite la boucle de la simulation. Le temps est découpé de manière discrète, en unité appelée pas de temps, qui représente une seconde de temps simulé.

Le deuxième élément est le planificateur d'agents, que nous appelons contexte. Le contexte est appelé à chaque pas de temps depuis la boucle principale. Il va servir à activer un à un les agents présents, en leur faisant exécuter leur procédure d'action.

Finalement, le troisième élément est la définition d'un agent, entité qui représente un voyageur. Un procédure d'action est implémentée, qui est exécutée à chaque pas de temps.

3.3 Représentation de l'environnement

Les agents évoluent dans un environnement représentant le réseau de trafic routier. Celui-ci est constitué de routes qui se croisent, sur lesquelles les agents vont pouvoir évoluer. La manière la plus simple pour modéliser ce réseau routier est la structure de graphe. Dans les quelques paragraphes qui suivent, nous rappelons quelques notions sur la théorie de graphe, que nous réutilisons tout au long de ce chapitre.

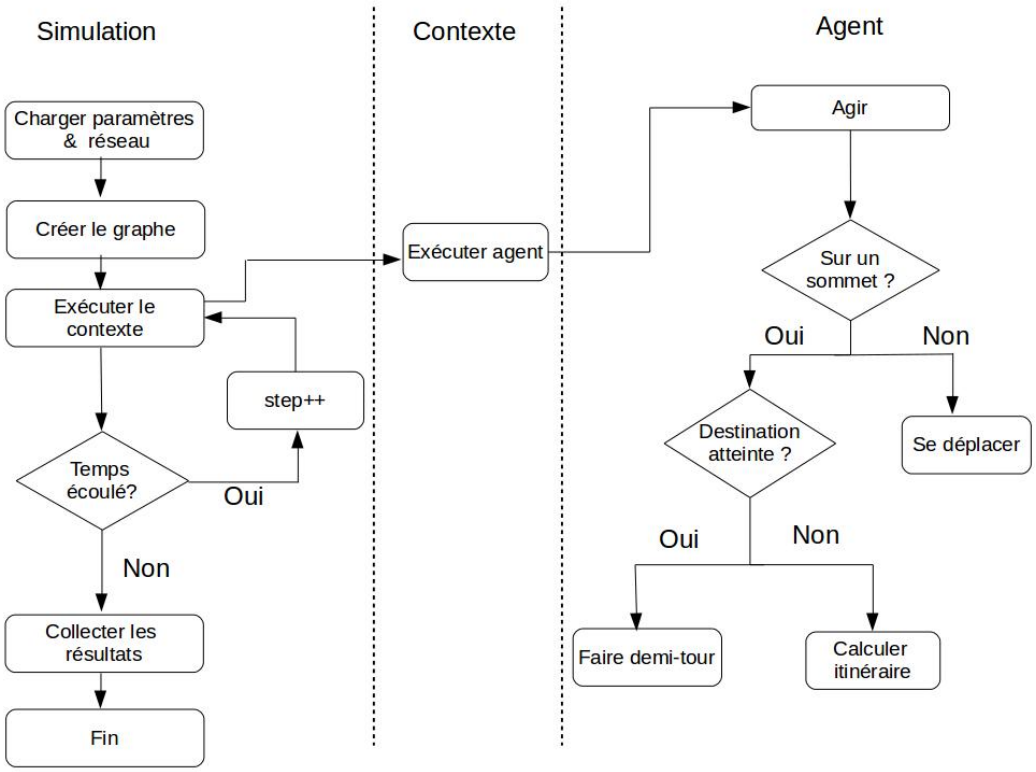

Figure 3.1: Fonctionnement de la simulation

3.3.1 Éléments de théorie des graphes

Un graphe G = (V,E) est défini par l'ensemble de ses sommets, notés $V = v1, v2, ..., vn$ et de ses arêtes, notées $E = e1, e2, ..., em$. Une arête e de l'ensemble E est définie de manière unique par une paire de sommets non ordonnée que l'on appelle les extrémités de e. Deux sommets reliés par une arête sont dits adjacents, et l'arête les reliant est dite incidente à ces deux sommets. On appelle ordre d'un graphe le nombre de sommets n de ce graphe. On appelle le degré d'un sommet le nombre de ses voisins.

Un graphe est complet si chaque sommet du graphe est relié directement à tous les autres sommets (figure 3.2).

3.3.1.1 Chemins et connexité

Un chemin menant du sommet a au sommet b est une suite de sommets, commençant par a et se terminant par b, de sorte que chaque sommet le composant soit reliés deux à deux par une arêtes. Par exemple, la figure 3.3 nous montre un chemin $(6, 3, 1, 2)$ allant

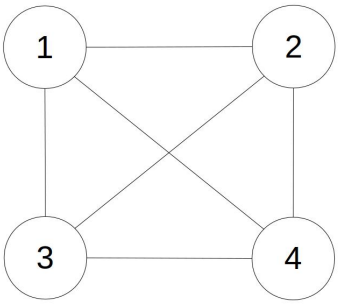

Figure 3.2: Un exemple de graphe complet

du sommet 6 au sommet 2.

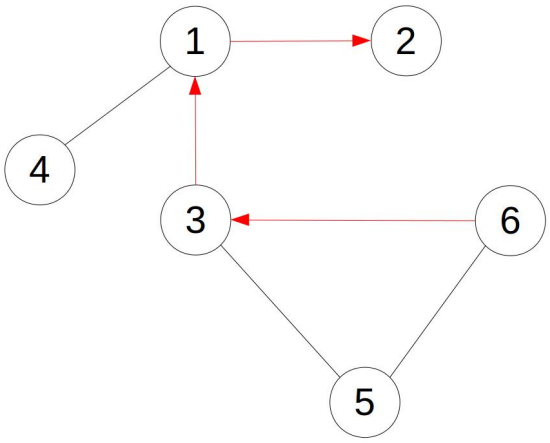

Figure 3.3: Un chemin dans un graphe

Un graphe connexe signifie qu'à partir de n'importe quel sommet, il existe un chemin au sein du graphe permettant de rejoindre n'importe quel autre somment. Un graphe non connexe se décompose en composantes connexes. Sur le graphe montré en figure 3.4, les composantes connexes sont {1,3,4,5,6} et {2,7,8}.

3.3.1.2 Graphes pondérés

Un graphe pondéré est un graphe étiqueté où chaque arête est affectée d'un nombre réel positif, appelé poids de cette arête. Le poids d'un chemin est la somme des poids des arêtes qui la composent. La distance entre deux sommets d'un graphe est le poids du

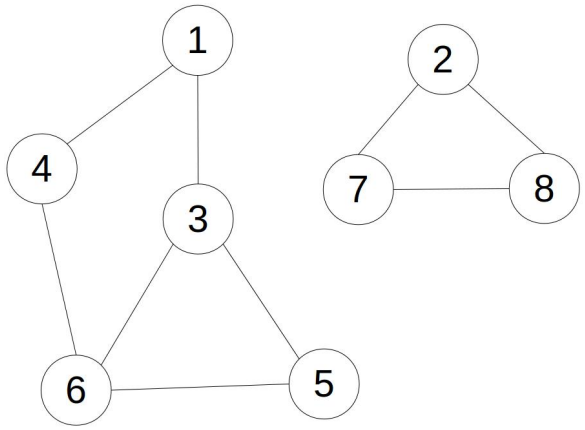

Figure 3.4: Un graphe possédant deux composantes connexes

plus court chemin les reliant. S'il n'existe pas de chemin entre les sommets a et b, alors $d(a, b) = \infty$. Par exemple, sur le graphe pondéré de la figure 3.5, $d(5, 4) = 13$, $d(3, 7) = \infty$, et $d(8, 7) = 2$. Un circuit, ou un cycle, est un chemin dont le sommet de départ et d'arrivée est le même.

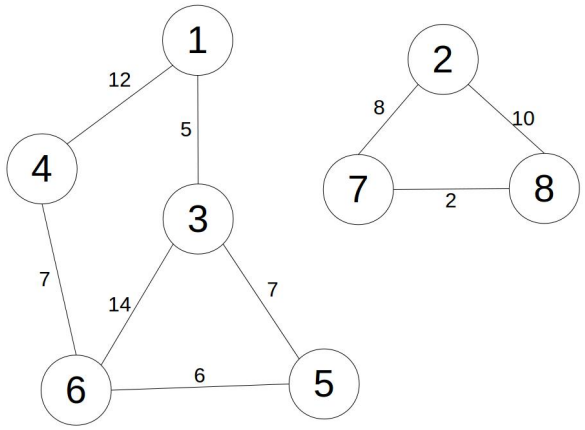

Figure 3.5: Un graphe pondéré

3.3.1.3 Graphes orientés

Un graphe dont les arêtes ne peuvent être empruntées que dans un sens est un graphe orienté. Il est défini par l'ensemble de ses sommets $V = v1, v2, ..., vn$ ainsi que par l'ensemble

de ses arcs $E = e1, e2, ..., em$. Un arc e est une paire ordonnée de sommets. Ainsi $e = (u, v)$ signifie que l'arc e part du somment o et arrive au sommet d. Ces deux sommets sont aussi respectivement appelés origine et destination de l'arc. Soit v un sommet d'un graphe orienté. On note $d^+(v)$ le degré extérieur du sommet v, c'est-à -dire le nombre d'arcs ayant v comme origine. On note $d^-(v)$ le degré intérieur du sommet v, c'est-à -dire le nombre d'arcs ayant v comme destination. Le degré d'un sommet v d'un graphe orienté est défini comme suit : $d(v) = d^+(v) + d^-(v)$.

3.3.1.4 Arbres

Un arbre est un graphe non orienté, acyclique et connexe. Un arbre couvrant d'un graphe non orienté et connexe est un arbre inclus dans ce graphe et qui connecte tous les sommets du graphe. Un graphe orienté est un arbre enraciné de racine v si et seulement si :

1. il est connexe,
2. v est l'unique sommet sans prédécesseur,
3. tous ses autres sommets ont exactement un prédécesseur.

3.3.2 Modélisation du réseau de transport

L'environnement multi-agent d'une simulation de mobilité est constitué du réseau de transport dans lequel les agents voyageurs évoluent. Nous modélisons ce réseau de transport à l'aide d'un graphe orienté $G(V, E)$ où $E = \{e_1, ..., e_n\}$ est un ensemble d'arcs représentant les routes et $V = \{v_1, ..., v_n\}$ est un ensemble de sommets représentant les intersections.

Un ensemble d'agents A se déplace dans ce réseau depuis des origines vers des destinations en essayant de minimiser leur temps de parcours. Le temps de parcours d'un arc au temps t dépend de sa longueur et du nombre d'agents présents sur cet arc à ce moment t.

C'est pourquoi les arcs et les sommets sont tous les deux valués par des entiers positifs évoluant dynamiquement avec le nombre d'agents présents. On note $|A_v|$ le nombre d'agents présents sur le sommet v et $|A_e|$ le nombre d'agents présents sur l'arc e. $|A_V| = \sum_{v \in V} |A_v|$ est le nombre d'agents présents sur le sous-ensemble de sommets V et $|A_E| = \sum_{e \in E} |A_e|$ est le nombre d'agents présents sur le sous-ensemble d'arcs E. Les arcs sont doublement valués, par le nombre d'agents présents d'une part, qui évolue dynamiquement, et par leur longueur d'autre part, qui elle ne change pas et est définie statiquement. Le coût de traversée d'un arc représente le temps qu'un agent va mettre pour le parcourir. Cette valeur sera celle utilisée pour les calculs de plus courts chemins par les agents. Elle est définie différemment en fonction du modèle de la simulation.

3.4 Comportement des agents

Lors de leur création, on affecte aux agents un nœud d'origine o et un nœud de destination d. Nous considérons que les agents vont tenter de minimiser leurs temps de parcours. La première action qu'un agent exécute lorsqu'ils est activé pour la première fois par son contexte est de calculer un plus court chemin entre o et d sur le graphe G qui évolue dynamiquement avec le trafic. A chaque pas de temps, il se déplace le long de son arc, à une vitesse calculée selon le modèle de la simulation.

Avec le guidage de type GPS, qui est généralisé depuis la démocratisation des smartphones, on considère que les usagers du réseau, et donc les agents les représentant, disposent d'une information en temps réel sur l'état du trafic. Si des congestions se produisent sur une portion de route que prévoyait de prendre un agent, cet agent devra modifier l'itinéraire prévu. C'est pourquoi, lorsqu'un agent atteint un nouveau nœud, il vérifie si les conditions de trafic ont évolué, en effectuant un nouveau calcul d'itinéraire sur l'état actuel du réseau. Comme les demi-tours sont impossibles (le graphe est orienté), il n'est pas nécessaire d'effectuer ce calcul lorsqu'un agent se déplace sur un arc.

Il est ainsi nécessaire de définir deux comportements pour les agents : comment choisissent-ils leur itinéraire d'une part, et comment se déplacent-ils dans le réseau d'autre part. Pour répondre à la problématique du choix de l'itinéraire, nous choisissons de modéliser le réseau sous la forme d'un graphe pondéré sur lequel nous utilisons des algorithmes de plus courts chemins. La deuxième problématique est dépendante du choix du modèle (microscopique ou macroscopique). Avec le modèle macroscopique, la vitesse est déduite d'équations de flux, tandis que nous nous intéressons aux micro-interactions entre les agents avec le modèle microscopique.

3.4.1 Calcul de l'itinéraire

La première fois qu'il est activé, un agent calcule un itinéraire au sein du graphe représentant le réseau de transport pour se rendre de son emplacement actuel (son point d'origine) à sa destination. A chaque sommet visité, il recalcule son itinéraire, afin de vérifier si les conditions de navigation n'ont pas changées. En effet peut être qu'un incident a eu lieu sur une partie du réseau qu'il comptait emprunter, et il existe alors un nouveau chemin plus court le conduisant à sa destination, suivant la dynamique du réseau (le nombre d'agents par arc à ce moment là).

Les calculs d'itinéraires sont effectués en utilisant l'algorithme de Dijkstra [28], sur la base de l'état actuel du réseau. Cet algorithme, proposé par Edgser Wybe Dijkstra en 1959, permet de calculer le plus court chemin entre un sommet particulier et tous les autres. Le résultat est un arbre couvrant le graphe, ayant pour racine le sommet d'origine.

L'algorithme de Dijkstra est présenté en pseudocode dans l'algorithme 1. L'arbre couvrant

de racine v_{deb} est codé à l'aide de deux tableaux : $dist[]$, représentant la distance entre v_{deb} et chacun des sommets, et $pred[]$, représentant le prédécesseur de chaque sommet dans l'arbre couvrant. On commence par initialiser tous les sommets avec une distance infinie, et un prédécesseur nul. On itère ensuite de manière gloutonne sur l'ensemble des sommets, choisissant le sommet u parmi ceux qui n'ont pas encore été traités qui minimise la distance avec v_{deb}. Pour chaque voisin de ce sommet, on met à jour les informations de l'arbre les concernant si le chemin passant par u pour les atteindre est le plus court.

Algorithm 1 Algorithme de *Dijkstra*

Pré-conditions :graphe $G = (V, E)$, $v_{deb} \in V$
Post-conditions :un arbre couvrant le graphe G
créer Q un ensemble de sommets
pour $v \in V$ **faire**
  $dist[v] \leftarrow \infty$
  $pred[v] \leftarrow \emptyset$
  ajouter v à Q
fin pour
$dist[v_{deb}] \leftarrow Q$
tant que $Q \neq \emptyset$ **faire**
  $u \leftarrow$ sommet de Q avec $dist[u]$ minimal
  enlever u de Q
  $N \leftarrow$ ensemble de voisins de u
  pour $v \in N$ **faire**
    $d \leftarrow dist[u] + distance(u, v)$
    si $d \leq dist[v]$ **alors**
      $dist[v] \leftarrow d$
      $prev[v] \leftarrow u$
    fin si
  fin pour
fin tant que
retourner $dist[], pred[]$

Si l'on est seulement intéressé par le plus court chemin entre une origine et une destination unique, on peut arrêter la recherche lorsque l'on atteint cette destination. Lorsque l'on dispose de l'arbre couvrant, on obtient le chemin de l'origine à la destination par itération inverse, en commençant par la destination et en remontant l'arbre jusqu'à l'origine. Cet algorithme est de complexité polynomiale : $O(|E| + |V|^2)$, c'est à dire $= O(|V|^2)$.

3.4.2 Déplacements dans le réseau

A chaque pas de temps de la simulation, si les agents se trouvent sur un arc, ils avancent sur cet arc autant que possible. C'est à dire en avançant aussi vite que le permettent les conditions de circulation sur l'arc qu'ils empruntent actuellement. Leur vitesse est calculée différemment en fonction du modèle de simulation choisi. Le mode de calcul utilisé pour

chaque modèle est détaillé dans les parties suivantes.

Lorsqu'un agent atteint le sommet se trouvant à la fin de son arc, il s'arrête un pas de temps pour effectuer un nouveau calcul d'itinéraire. Au pas de temps suivant, il s'engagera sur le prochain arc de son itinéraire.

Lorsqu'un agent a atteint sa destination, il s'arrête pendant un pas de temps, puis repart vers son point de départ. Nous avons choisi ce mode opératoire afin de garder un nombre constant d'agents présents dans la simulation. La simulation se termine lorsqu'un certain nombre (défini en paramètre) de pas de temps est écoulé.

3.4.3 Spécificités de la simulation macroscopique

Dans la simulation de trafic macroscopique, on ne s'intéresse pas aux détails de chaque portion de route, mais on envisage plutôt une modélisation d'ensemble, sous forme de flux. Dans notre simulateur dans sa version macroscopique, bien que les agents vont prendre leurs décisions individuellement, ils ne vont pas s'occuper de l'état précis de l'environnement les entourant. Ils vont calculer leur vitesse d'évolution au sein d'un arc en fonction de l'état global du flux de véhicules sur cet arc.

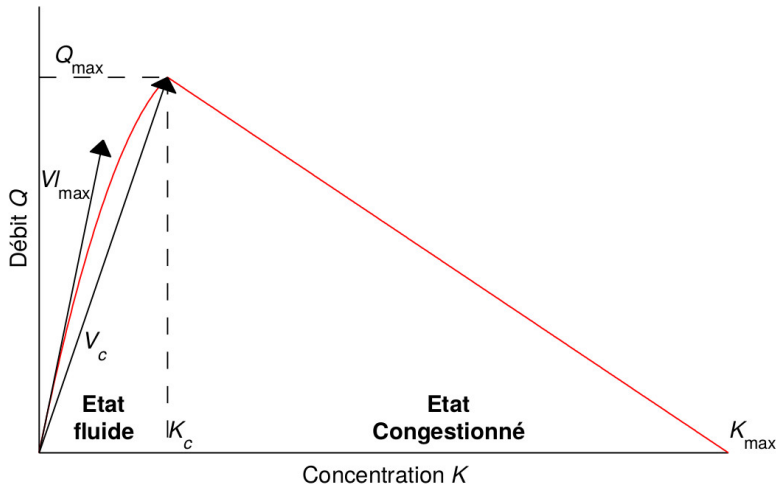

Figure 3.6: Illustration du digramme fondamental du trafic routier

Nous utilisons le diagramme fondamental triangulaire [38] du trafic qui donne une relation entre le débit Q (véhicules/heure) et la densité K (véhicules/km) (figure 3.6). Le diagramme fondamental spécifie que plus l'on dépasse une densité critique de véhicules K_c sur une portion de route, plus les véhicules empruntant cette route sont ralentis. Voici

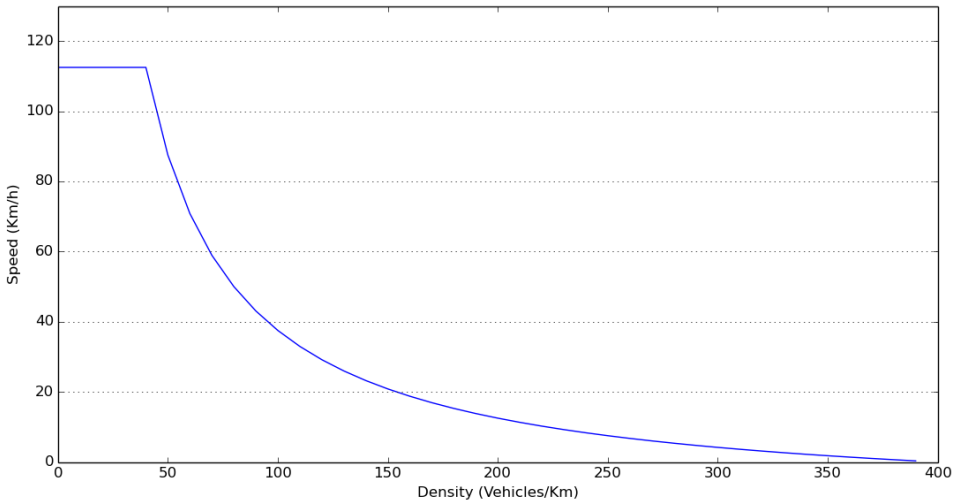

Figure 3.7: Vitesse en fonction de la densité

l'équation que nous utilisons pour modéliser ce phénomène :

$$Q = \begin{cases} \alpha K & \text{if} \quad K \leq K_c \\ -\beta(K - K_c) + \alpha K_c & \text{if} \quad K > K_c \end{cases} \tag{3.1}$$

Cette équation est paramétrée par α la vitesse en régime fluide, β la vitesse de propagation de la congestion et K_c la densité critique. Comme $v = \frac{Q}{K}$, on a :

$$v = \begin{cases} \alpha & \text{if} \quad K \leq K_c \\ \frac{-\beta(K-K_c)+\alpha K_c}{K} & \text{if} \quad K > K_c \end{cases} \tag{3.2}$$

Ainsi nous pouvons définir une fonction de coût qui retourne un temps de parcours par unité de longueur $(1/v)$ en fonction de A_e, le nombre d'agents présents sur un arc (figure 3.7).

$$cost(|A_e|) = \begin{cases} \frac{1}{\alpha} & \text{if} \quad |A_e| \leq K_c \\ \frac{|A_e|}{-\beta(|A_e|-K_c)+\alpha K_c} & \text{if} \quad |A_e| > K_c \end{cases} \tag{3.3}$$

A chaque pas de temps, les agents évoluant sur un arc vont calculer combien ils avancent grâce à cette fonction de coût, qui prend en paramètres le nombre d'agents présents sur l'arc, ainsi que les caractéristiques de la route (α, β et K_c). Le calcul des plus courts chemins est aussi effectué grâce au diagramme fondamental. En effet, la pondération d d'un arc e est mise à jour à chaque pas de temps de la manière suivante : $d_e = \frac{cost(|A_e|)}{Longueur_e}$. La détermination exacte des paramètres de la route, nécessaires pour une simulation

réaliste, demande des données réelles et un processus de calibration itératif. Cependant, une simulation réaliste est au-delà des ambitions de ce simulateur, qui est, rappelons le, destiné à mesurer l'efficacité des méthodes de parallélisation. Nous avons ainsi affecté à chacune des portions de route du réseau les paramètres suivant : $\alpha = 2$, $\beta = 2.3$ et K_c =longueur de la route en mètre (plus la route est longue, plus sa densité critique sera grande).

3.4.4 Spécificités de la simulation microscopique

Comme nous l'avons vu dans le chapitre 1, de nombreux simulateurs de trafic multi-agents de la littérature implémentent des comportements microscopiques pour modéliser les mouvements des agents. L'information dont les agents se servent pour adapter leur vitesse se trouve uniquement dans leur environnement local. Ils vont percevoir une partie de leur environnement direct, c'est à dire les agents qui les entourent, pour calculer leur prochain mouvement. Contrairement au modèle macroscopique, ce paradigme implique de nombreuses interactions directes entre les agents. L'action de chacun des agents est en effet conditionnée par les actions des autres agents présents dans leur zone de perception.

Pour représenter cette topologie de communication, notre simulateur, dans sa version microscopique, implémente un modèle de poursuite [36], qui est largement utilisé dans la simulation de trafic microscopique. Notre but n'étant pas d'obtenir un simulateur destiné à fonctionner dans un contexte opérationnel, nous avons choisi un modèle de poursuite assez simple. Les actions d'un agent donné sont influencées uniquement par l'agent le précédent directement sur l'arc qu'il emprunte. Nous n'implémentons pas de procédure de changement de voie, ni ne détaillons les différents types d'intersections. Le modèle de poursuite suffit à générer le grand nombre de communications locales nécessaires pour représenter les simulateurs microscopiques.

Lorsqu'un agent en suit un autre, on l'appelle agent suiveur, et l'agent leader l'agent qui est suivi. Lorsqu'un agent entre sur un arc, on lui assigne le dernier agent arrivé avant lui comme prédécesseur. A chaque pas de temps l'agent suiveur adapte sa vitesse en fonction de la vitesse et de la position du véhicule qu'il suit. Ce sont les différentiels de vitesse et de positions qui nous intéressent ici.

La distance séparant le véhicule n suiveur du véhicule leader est appelée distance inter-véhiculaire. C'est la distance entre l'avant du véhicule leader et l'avant du véhicule suiveur et elle est notée $s_n(t)$ (figure 3.8). Le temps inter-véhiculaire, est le temps nécessaire à garder entre deux véhicules par unité de vitesse. Le différentiel de vitesse est appelée vitesse relative. Le temps de réaction du conducteur, noté T est pris en compte. Nous considérons ici qu'il est de une seconde (un pas de temps de la simulation). Les variables nécessaires pour décrire notre modèle sont les suivantes :

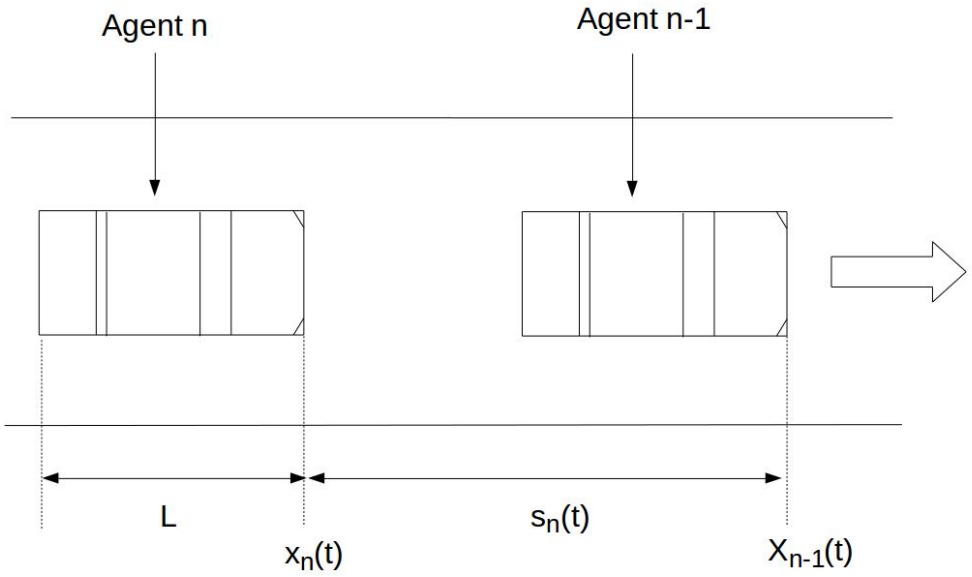

Figure 3.8: Le modèle de poursuite

$x_n(t)$	position de l'agent n au temps t
$x'_n(t)$	vitesse de n au temps t
$x''_n(t)$	accélération de n au temps t
$s_n(t) = x_{n-1}(t) - x_n(t)$	distance inter-véhiculaire
$s'_n(t) = x'_{n-1}(t) - x'_n(t)$	vitesse relative
T	temps de réaction
h	temps inter-véhiculaire

A chaque pas de temps, la vitesse d'un agent est calculée grâce à la relation suivante :

$$x''_n(t+T) = \lambda s'_n(t) + \sigma(s_n(t) - d_n(t)) \tag{3.4}$$

Avec :

$$d_n(t) = L + hx'_n(t) \tag{3.5}$$

Si personne ne précède un agent donné, celui-ci accélère au maximum de ses capacités jusqu'à ce qu'il atteigne la limitation de vitesse propre à la portion de route sur laquelle il circule. Dans notre implémentation du modèle, nous utilisons les valeurs suivantes :

- $\lambda = 0,4s^{-1}$
- $\sigma = 0,14s^{-2}$
- $T = 1s$
- $h = 1,8s$

Pour le calcul de leur itinéraire, les agents ne disposent pas dans ce modèle de l'information de temps de parcours de chaque arc fournie dans le modèle macroscopique par le diagramme fondamental. Nous avons ainsi implémenté un mécanisme permettant de fournir aux agents des temps de parcours actualisés en fonction de l'état du trafic. A chaque fois qu'un agent atteint le sommet terminant l'arc courant, l'arc va enregistrer combien de temps il lui a fallu pour le traverser. Cette information donne une estimation constamment mise à jour du temps requis pour la traversée. Ce temps de parcours dynamique est utilisé comme valeur de coût des arcs par les agents lors de leur exécution de l'algorithme de Dijkstra.

3.5 Implémentation du simulateur

Nous avons choisi d'implémenter notre simulateur avec le langage Python, pour son efficacité dans le prototypage rapide. Python est un langage portable, mature, disposant de nombreuses bibliothèques scientifiques éprouvées. Il est avec C et Fortran l'un des langages les plus utilisés pour le calcul haute performance [49]. Comme le but de nos travaux est de fournir une étude de l'efficacité relative de différentes méthodes de distribution, et non une performance absolue, le choix de Python nous semble pertinent.

Pour la modélisation du réseau de trafic proprement dit, nous utilisons igraph, une collection d'outils d'analyse de réseau *open source*, qui dispose d'une interface Python. Cette bibliothèque nous sert à représenter le réseau sous la forme d'un graphe orienté. Elle permet le multi-étiquetage des arcs et des sommets. Elle est capable de charger des graphes à partir de fichiers les décrivant, de les afficher, ainsi que d'y effectuer divers traitements. Elle propose par exemple une implémentation optimisée de l'algorithme de Dijkstra, ainsi que des modèles de création de graphes aléatoires.

Nous avons divisé notre programme en trois classes python (figure 3.9). La classe principale, nommée *Simulation*, prends les paramètres de la simulation, charge le graphe, crée les agents aléatoirement et gère le décompte du temps. La classe *Context* gère les agents et fait appel à eux à chaque pas de temps. Enfin la classe *Agent* gère le comportement des agents, en implémentant leur procédure de calcul de chemin et de déplacement.

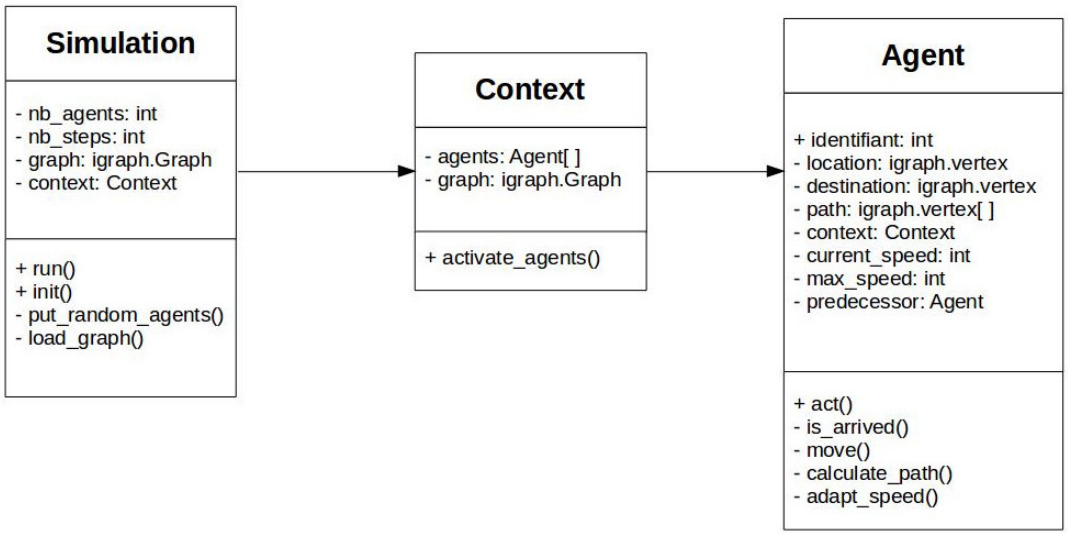

Figure 3.9: Diagramme UML du simulateur

3.6 Conclusion

Afin de tester nos méthodes de distribution de simulateurs de mobilité, il était nécessaire de disposer d'un environnement expérimental. C'est pourquoi nous avons développé un simulateur de mobilité qui garde les propriétés générales de tout simulateur de trafic multi-agent : des agents évoluent dans un réseau et cherchent à minimiser leur temps de parcours.

Il n'est pas aussi complexe que d'autres simulateurs de la littérature dont la vocation est d'obtenir une simulation réaliste en termes de trafic généré. Notre objectif est de ne représenter que les caractéristiques partagées par les simulateurs de trafic qui influent sur la complexité et l'efficacité du processus de distribution.

Les éléments gérés par notre simulateur sont le découpage et l'écoulement du temps, la modélisation du réseau de transport, le calcul d'itinéraire des agents, et le déplacement des agents. Ce dernier point est le seul qui diffère réellement entre les modèles microscopique et macroscopique. Le mode de communication inter-agent est en effet différent dans ces deux modèles.

Chapitre 4

Méthodes de distribution

Sommaire

4.1 Introduction

Pour lancer notre simulation à l'échelle d'une ville, nous avons besoin d'une capacité de mémoire et d'une puissance de calcul importante. C'est pourquoi nous cherchons à

déployer la simulation sur plusieurs hôtes. Une telle simulation s'exécutant sur un cluster est typiquement de type SPMD[1], qui est une sous-catégorie du mode de fonctionnement MIMD de la taxonomie de Flynn. Dans le mode SPMD, tous les processeurs, reliés par un réseau, exécutent le même programme global, répliqué sur des mémoires indépendantes. Chaque processeur n'a accès qu'à une partie des données dans sa mémoire privée. Ils vont en outre exécuter seulement une partie du programme global, en fonction de leur rang. L'avantage de cette approche est sa grande évolutivité; elle peut être mise en œuvre sur la plupart des architectures parallèles et nous pouvons déployer la même simulation sur des systèmes plus puissants si nécessaire.

4.2 Le découpage de la simulation

Pour que la distribution soit efficace, il est nécessaire de répartir la charge de travail le plus équitablement possible entre les unités de calcul que l'on a à notre disposition. Des disparités dans la répartition peuvent en effet avoir des conséquences sur le temps d'exécution global de la simulation. Le temps d'exécution de la simulation distribuée sera quoi qu'il arrive supérieur ou égal au temps mis par l'UC la plus lente. La performance individuelle d'une UC va décroître d'autant que sa charge va dépasser sa capacité nominale. D'autre part, si l'une des UC a très peu de calculs a effectuer, elle aura terminé plus vite que les autres, et la puissance de calcul qui aurait pu être utilisée pour réduire le temps de la simulation sera perdue.

Rihawi et al. [72–75] ont proposé des mécanismes permettant de distribuer la charge d'une simulation agent-situé sur un espace euclidien. Les résultats qu'ils ont obtenus sont prometteurs pour certains modèles. Nous nous inspirons de leurs propositions pour les adapter aux spécificités d'une simulation de mobilité, qui modélise des voyageurs se déplaçant sur un réseau de transport.

Dans notre modèle, la charge de travail principale est générée par les calculs de plus courts chemins effectués par les agents quand ils atteignent une nouvelle intersection, ainsi que par leur adaptation aux agents environnant (dans le modèle microscopique). Ainsi, si trop d'agents sont concentrés sur une seule UC, celle-ci va devenir le goulot d'étranglement de la simulation et les performances du système vont décroître.

Le paradigme multi-agents offre par sa nature même une unité de découpage de la quantité de calculs, qui est l'agent lui même. Par conséquent, afin de distribuer ce modèle, nous devons partager le plus uniformément possible les agents entre les hôtes.

[1] *Single Program, Multiple Data*

4.2.1 Synchronisations entre les unités de calcul

Lorsque l'on distribue une simulation entre plusieurs UC, chaque UC gère un sous-ensemble de la simulation. Pour que celle ci reste cohérente dans son ensemble, il est indispensable de mettre en place des mécanismes permettant aux UC de se synchroniser. Comme l'explique Fujimoto [35], il existe deux types de mécanismes de synchronisation inter-UC dans une simulation distribuée : conservateur et optimiste.

Avec l'approche conservatrice, toutes les informations nécessaires à la cohérence globale de la simulation sont échangées par les UC avant qu'elles ne commencent à effectuer le moindre calcul. Au contraire, avec l'approche optimiste, les UC tolèrent que la simulation soit dans un état inconsistant afin de prendre de l'avance dans les calculs. Ces derniers sont en effet effectués sans attendre que l'intégralité des informations soient échangées, quitte à revenir en arrière pour redonner un état consistant à la simulation.

Le choix du type de synchronisation dépendra fortement du modèle implémenté. En effet, dans un modèle où peu d'échanges d'informations sont nécessaires pour garder un état cohérent, il peut être intéressant d'utiliser un mode de synchronisation optimiste, quitte à perdre ensuite un peu de temps pour la correction d'erreurs. Par contre, dans les simulations où chaque sous-ensemble dépend très fortement de l'état d'exécution des autres sous-ensembles, le mode optimiste peut se révéler très coûteux en termes de temps de calcul et de consommation mémoire.

Dans une simulation de trafic multi-agent, les actions des agents vont être largement dépendantes de l'état global de la simulation (ne serait-ce que pour leurs calculs de chemins). C'est pourquoi nous avons choisi d'utiliser le mode de synchronisation conservateur pour la distribution de notre simulateur : les informations nécessaires au déroulement de la simulation seront échangées à la fin de chaque pas de temps, et le pas de temps suivant ne débutera pas avant que cet échange ne soit finalisé.

4.2.2 Le problème de la répartition optimale

Notre problème consiste donc en la répartition de n agents sur k unités de calcul, qui sont connectées entre elles. Nous partons ici du principe que chaque UC dispose de la même puissance de calcul. La quantité de travail d'une UC à un instant donné est générée par les agents présents sur cette unité à cet instant. Ainsi un agent peut être considéré comme une unité atomique de travail. Soit w_i le charge de travail générée par l'agent a_i. La charge de l'UC U_i, notée $W(U_i)$ est donc la suivante :

$$W(U_i) = \sum_{a_j \in U_i} w_j \tag{4.1}$$

Les interaction entre les agents présents sur une même unité de calcul seront moins

coûteuses que celles se déroulant entre des agents présents sur différentes UC. Ce dernier cas engendre en effet des messages qui devront transiter sur le réseau de communication. Définissons $p(a_i, a_j)$ le coût des interactions entre les agents a_i et a_j. Si ils se trouvent sur la même UC, ce coût est nul. Le coût de communication entre deux UC est de :

$$P(U_i, U_j) = \sum_{a_k \in U_i} \sum_{a_l \in U_j} p(a_k, a_l) \tag{4.2}$$

Une répartition idéale de n agents sur k unités de calcul est une optimisation multi-objectif qui doit :

- Distribuer la charge de travail au mieux entre les UC,
- Limiter autant que possible les communications inter-UC.

C'est à dire, de manière formelle, trouver la répartition qui minimise:

$$\sum_{1 \leq i \leq j \leq k}^{i \neq j} P(U_i, U_j) \text{ avec } W(U_i) = \frac{n}{k} + \epsilon \tag{4.3}$$

Avec ϵ le plus petit possible:

Certains travaux, comme [65], effectuent un optimisation linéaire du problème en nombres entiers. Bien que cette approche permet de trouver une solution optimale au problème de la répartition, la complexité des calculs augmente de façon exponentielle avec le nombre d'agents. Les simulations de trafic à large échelle impliquent, comme leur nom l'indique, un très grand nombre d'agents. Il n'est donc pas envisageable d'utiliser une optimisation linéaire.

Nous disposons de deux approches pour répartir la simulation [82],[74]. L'une d'elles est de faire totalement abstraction des coûts de communication, et de se concentrer uniquement sur le critère de la charge de travail, en distribuant les agents le plus équitablement possible entre les UC. L'autre est une heuristique basée sur l'environnement : elle prend comme hypothèse que les agents se situant le plus proche dans leur environnement auront une probabilité plus grande de communiquer entre eux. Le monde virtuel va alors être découpé, et chaque UC sera en charge d'une partie seulement de l'environnement de la simulation, ainsi que des agents qui la peuplent.

De nombreuses propositions de découpage de l'environnement ont été faites dans la littérature, comme par exemple [66], [69] ou [51]. Ces travaux présentent des méthodes permettant le découpage de l'espace euclidien de la simulation, afin de répartir les parties du plan entre les UC disponibles. Ces techniques sont particulièrement utilisées par les jeux vidéo massivement multi-joueurs, afin de répartir l'univers virtuel, et les joueurs qui le peuplent entre différents serveurs.

Dans notre simulateur de mobilité, nous avons modélisé l'environnement sous la forme d'un graphe. Les communications entre le agents se font de manière différente en fonction du modèle utilisé. Dans le modèle macroscopique, les agents n'interagissent pas directement, le passage d'information se fait via les contextes qui échangent le nombre d'agents présents sur leurs arcs. Dans le modèle microscopique, les agents vont interagir avec les agents proches d'eux dans le réseau de transport.

Bien qu'il soit toujours possible de projeter ce réseau sur un plan, puis de découper le plan selon les méthodes traditionnelles, il est plus efficace de tirer partie de la topologie particulière de chaque réseau de transport. Si l'on répartit le graphe sur plusieurs UC, les principales communications réseau générée par les agents seront lorsqu'ils devront être transférés d'une UC à l'autre, parce qu'ils se déplacent sur une partie du réseau qui n'est plus gérée par leur UC actuelle.

4.2.3 Le partitionnement de graphes

Nous avons choisi pour distribuer notre simulation, de découper le graphe représentant le réseau routier, puis de répartir les sous-graphes obtenus entre les unités de calcul.

Soit $G(V, E)$ un graphe. Un ensemble $P = V_1, V_2, ..., V_k$ de sous-ensembles de V est une partition de G si et seulement si :

1. Tous les sommets du graphe sont présents dans la partition :

$$\cup_{i=1}^{k} V_i = V \tag{4.4}$$

2. les sous-ensembles de la partition sont disjoints deux à deux :

$$\forall i, j \in \{1, ..., k\}^2, i \neq j, V_i \cap V_j = \emptyset \tag{4.5}$$

3. aucun élément de la partition n'est vide :

$$\forall i \in \{1, ..., k\}, V_k \neq \emptyset \tag{4.6}$$

4.2.3.1 Le poids de coupe

Afin de répartir au mieux les opérations de calcul, le nombre d'agents présent sur chacun des sous-graphes devra être équilibré entre les UC. De plus, la coupe devra être telle qu'elle minimise le nombre d'agents qui devra être transféré entre les unités de calcul.

Nous devons trouver une fonction de poids $w(u, v)$ qui représente à quelle point il est coûteux (en termes de futurs transferts d'agents d'une UC à une autre) de couper l'arc (u, v). Nous pourrons ainsi définir le poids (qu'il faudra minimiser) de coupe de deux

sous-ensembles V_1 et V_2 de sommets d'un graphe $G(V, E)$ par :

$$\theta(V_1, V_2) = \sum_{u \in V_1, v \in V_2} w(u, v) \tag{4.7}$$

Ainsi le poids de coupe d'une partition du graphe sera de :

$$\theta(P) = \sum_{i \leq j} \theta(V_i, V_j) \tag{4.8}$$

On peut prendre comme fonction de poids $w(u, v) = 1$ pour tous les arcs. Cela reviendrait à définir notre poids de coupe comme étant le nombre d'arcs coupées lors de la partition. Moins nombreuses seront les connexions possibles entre les partitions, et moins nombreux seront les agents à transférer entre les UC.

Dans le cas d'une simulation réelle, une fonction de coût plus efficace peut être définie à partir de données mesurées sur le terrain : le poids d'un arc sera alors égal à la fréquentation moyenne d'une portion de route. Dans notre simulation, nous ne disposons pas de ce type de données. Comme nos agents ont des origines/destinations aléatoires, la notion de centralité intermédiaire peut être utile. La centralité intermédiaire d'un arc est égale au nombre de tous les plus courts chemins existants passant par cet arc. Soient u et $v \in V$, σ_{st} le nombre total de plus courts chemins allant du sommet s au sommet t et $\sigma_{st}(u, v)$ le nombre de ces chemins passant par l'arc (u, v). La centralité intermédiaire de (u, v) est définie comme suit :

$$g(u, v) = \sum_{s \neq u \neq v \neq t} \frac{\sigma_{st}(u, v)}{\sigma_{st}} \tag{4.9}$$

Nous pouvons définir grâce à la centralité intermédiaire une fonction de poids qui correspond à la fréquentation moyenne d'une arrête pour une simulation aux origines et destinations choisies aléatoirement.

$$w(u, v) = g(u, v) \tag{4.10}$$

4.2.3.2 Les algorithmes de partitionnement

Notre problème consiste à répartir une simulation faisant agir un ensemble d'agents A évoluant sur un graphe comportant n sommets parmi k unités de calcul, de telle sorte que chaque unité ait approximativement le même nombre d'agents, et que le poids de coupe soit minimal. Nous devons ainsi trouver une partition du graphe représentant le réseau routier ayant les propriétés suivantes :

- chaque partie contient au plus $(1 + \epsilon)\frac{|A|}{k}$,
- $\theta(P)$ est minimal.

Ce problème est similaire au problème connu sous le nom $(k, 1+\epsilon)$-*balanced partitioning problem*, dont l'objectif est de couper un graphe en minimisant le nombre d'arêtes sectionnées par le plan de coupe, tout en équilibrant le nombre de sommets présents dans chaque partition.

Le problème du partitionnement de graphe a été largement étudié dans la littérature. Comme démontré dans [17], il s'agit d'un problème d'optimisation combinatoire NP-complet. La recherche exhaustive d'une solution exacte, par le parcours systématique de toutes les solutions possibles n'est donc pas une option, car l'espace des solutions est trop important pour des graphes de grandes tailles.

C'est pourquoi certaines heuristiques ont été proposées pour résoudre ce problème dans un temps raisonnable. La méthode spectrale [29] a été beaucoup utilisée dans le passé, mais a progressivement été remplacée par la méthode de partitionnement multi-niveaux. Cette dernière, originellement créée pour améliorer les techniques existantes [7] a été reconnue comme étant une méthode très puissante, qui offre une vision plus globale des graphes que les techniques traditionnelles.

4.2.4 Le partitionnement multi-niveaux

Comme la complexité du problème de partitionnement dépend de la taille du graphe, l'idée du partitionnement multi-niveaux est de regrouper les sommets et de travailler avec des groupes de sommets plutôt qu'avec des sommets indépendants. Il s'agit d'une méthode qui se déroule en trois phases. La première, appelée contraction, consiste en une réduction itérative du graphe, jusqu'à obtenir un graphe de la taille souhaitée. La deuxième phase est la phase de partitionnement proprement dite du graphe réduit. Dans la troisième phase, la contraction est itérativement projetée sur les graphes intermédiaires obtenus lors de la première phase, jusqu'à obtenir une partition du graphe initial. Entre chaque phase de projection, il est nécessaire d'utiliser un algorithme d'affinage, destiné à améliorer la partition projetée en prenant en compte les spécificités locales du graphe de chaque niveau (figure 4.1).

Le partitionnement multi-niveaux a été formalisé dans un cadre générique par Walshaw [83]. L'algorithme 2 en donne sa structure générale. Les parties suivantes présentent en détail des exemples d'algorithmes pouvant être utilisés pour chacune des étapes.

4.2.4.1 Contraction

À chaque niveau, les sommets du graphe sont regroupés pour former un graphe contracté. Une arrête (u, v) est contractée en replaçant les sommets qui lui sont incidents par un unique sommet t de poids $w(t) = w(u) + w(v)$. Toutes les arrêtes (u, s) et (v, s) sont remplacées par (w, s), pour conserver les informations de connectivité (figure 4.2).

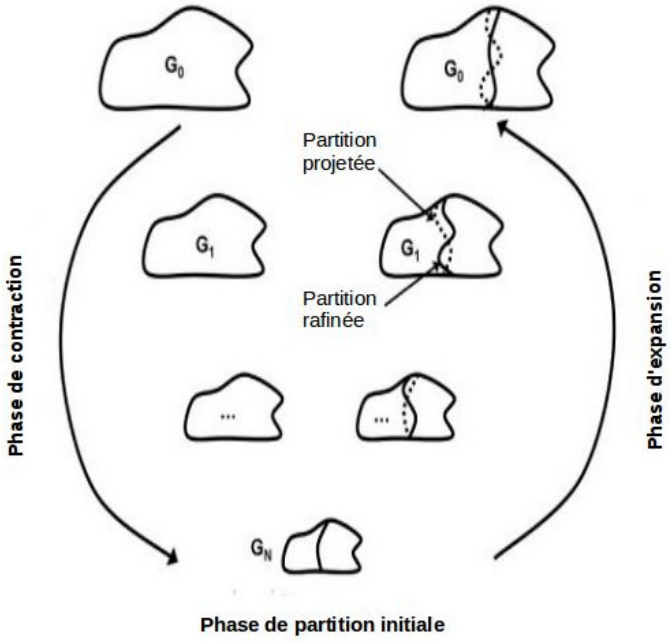

Figure 4.1: Les différentes phases du partitionnement multi-niveaux

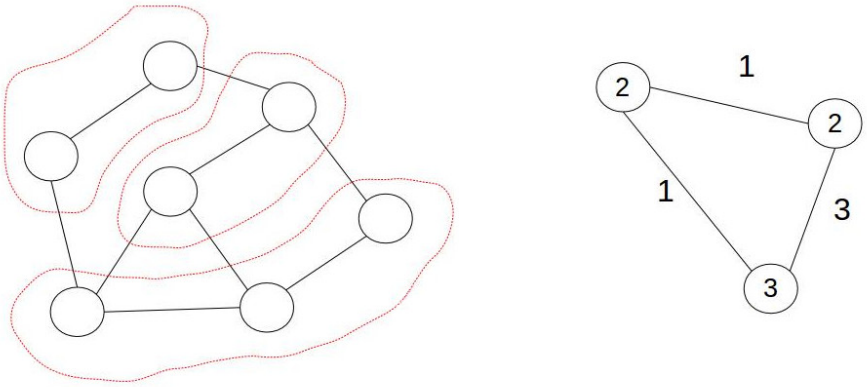

Figure 4.2: Un exemple de contraction de graphe

L'un des buts du partitionnement de graphe est de minimiser le coût de coupe. On peut donc intuitivement regrouper les sommets dont les arrêtes ont le poids maximum. Les arrêtes ayant un faible poids auront ainsi plus de chance de se faire couper. C'est l'approche utilisée par l'algorithme HEM [43], qui sélectionne aléatoirement un sommet du graphe, puis sélectionne l'arrête adjacente de poids maximum pour le supprimer et fusionner les deux sommets reliés par cette arrête.

Algorithm 2 Algorithme *Partitionnement multi-niveaux* (Walshaw)

Pré-conditions :$G = (V, E)$
Post-conditions : Partition P
$l \leftarrow 0$
tant que G_l trop grand **faire**
  $G_{l+1} \leftarrow contracter(G_l)$
  $l \leftarrow l + 1$
fin tant que
$P_l \leftarrow partitionner(G_l)$
tant que $l \geq 0$ **faire**
  $l \leftarrow l - 1$
  $P_l^0 \leftarrow projeter(P_{l+1}, G_l)$
  $P_l \leftarrow affiner(P_{l+1}^0, G_l)$
fin tant que
Retourner P_0

4.2.4.2 Partitionnement

Après les itérations successives des phases de contraction, nous obtenons un graphe de la taille souhaitée. Nous pouvons ainsi passer à la phase de partitionnement proprement dite. Les techniques les plus utilisées pour la partition du graphe contracté sont les algorithmes d'expansion de région. Ces derniers sont simples à implémenter, et sont particulièrement efficaces sur des problèmes de petite taille.

Les algorithmes existants suivent tous un même schéma général (algorithme 3). Ils commencent par affecter un sommet (le plus souvent choisi aléatoirement) à chaque partition. On ajoute ensuite itérativement des sommets aux partitions, jusqu'à ce que tous les sommets soient affectés. Le critère de sélection du sommet à ajouter (la fonction *choisir_sommet*() dans notre exemple) est le facteur discriminant les différents algorithmes appartenant à la classe des algorithmes d'expansion de région.

Un critère de sélection est de choisir aléatoirement un sommet parmi les sommets présents sur la frontière (ayant au moins une arrête commune avec l'un des sommets de la partition). Cette méthode est très rapide, mais fournit une partition qui ne tient pas en compte le nombre d'arêtes coupées par la partition. L'algorithme GGP[43] (*Graph Growing Partitionning*) utilise ce critère. Dans le même article, les auteurs proposent une version de l'algorithme qui prend en compte le gain de la coupe introduit par la sélection d'un sommet. Il s'agit de l'algorithme GGGP.

Cependant, dans la pratique, le critère de sélection de l'algorithme GGGP est insatisfaisant, car il génère de nombreuses situations d'égalité [9]. C'est pour pallier à ce problème que Battiti et Bertosi proposent le critère différentiel [8] (*Differential Greedy*), qui prend en compte la connectivité des sommets lors de leur sélection.

Soit le poids intérieur $int(v, p)$ le poids des connexions entre le sommet v et les sommets

Algorithm 3 Algorithme d'*Expansion de région*

Pré-conditions : Graphe $G = (V, E)$, taille k de la partition
Post-conditions : Partition P
$P \leftarrow P_0, ..., P_{k-1}$
$V' \leftarrow V$
pour $p \in [0, k-1]$ **faire**
  $v \leftarrow$ un sommet aléatoire de V'
  $P_p \leftarrow \{v\}$
  $V' \leftarrow V' \setminus \{v\}$
fin pour
tant que $|V'| > 0$ **faire**
  $v \leftarrow choisir\_sommet(V', P, p, G)$
  $P_p \leftarrow P_p \cup \{mv\}$
  $V' \leftarrow V' \setminus \{mv\}$
  $p \leftarrow (p+1) \bmod (k)$
fin tant que
Retourner P

de la partition p, et le poids extérieur $ext(v, p)$ le poids des connexions entre le sommet v et les sommets qui sont placés dans une autre partition que p.

$$int(v, p) = \sum_{u \in p} w(v, u) \tag{4.11}$$

$$ext(v, p) = \sum_{u \notin p} w(v, u) \tag{4.12}$$

L'algorithme DG (algorithme 4) sélectionne le sommet qui maximise la quantité suivante :

$$diff(v, p) = int(v, p) - ext(v, p) \tag{4.13}$$

Algorithm 4 Fonction de sélection de l'algorithme *Differential Greedy*

Pré-conditions : Une liste V' de sommets non assignés, l'ensemble des partitions P, la partition courante p, le graphe $G = (V, E)$
Post-conditions : Un sommet qui sera assigné à la partition P_p
$m = min_{v \in V'} diff(v, p)$
$S = \{v \in V' | diff(v, p) = m\}$
Retourner Un sommet aléatoire de S

Ainsi, l'algorithme DG prend efficacement en compte la structure du graphe à chaque itération de l'algorithme d'expansion, et fournit une partition de qualité. Il peut efficacement être mis en œuvre dans les graphes réduits que l'on est amené à traiter dans le partitionnement multi-niveaux.

4.2.4.3 Projection et affinage

La partition obtenu sur le graphe contracté pourrait être itérativement projetée sur les différents sous-graphes obtenus lors de la phase de contraction. Cependant, George Karypis et Vipin Kumar [44], prouvent qu'une partition optimale d'un graphe contracté n'est pas nécessairement optimale lorsqu'elle est projetée sur le graphe initial.

C'est pourquoi il est important d'affiner la partition projetée à chaque étape pour améliorer localement le coût de coupe. Brian Kernighan et Shen Lin ont proposé en 1970 un algorithme [46] permettant d'affiner un bissection d'un graphe en échangeant successivement deux sous-ensembles jusqu'à ce qu'aucune bissection plus performante ne puisse être trouvée.

Pour ce faire, les auteurs définissent la notion de gain d'échange, qui est une fonction mesurant l'utilité de l'échange de deux sommets entre les deux partitions. Soit deux partitions A et B et deux sommets $u \in A$ et $v \in B$:

$$g(u,v) = diff(u,B) + diff(v,A) - 2w(u,v) \tag{4.14}$$

L'objectif de l'algorithme KL (Kernighan-Lin) est de trouver une série optimale d'échange entre les deux partitions qui maximise g (algorithme 5). L'algorithme est d'autant plus efficace que sa partition initiale est de bonne qualité.

Algorithm 5 L'algorithme de Kernighan-Lin

Pré-conditions : Un graphe $G = (V, E)$, et deux partitions A et B de ce graphe
Post-conditions : Deux partitions A et B optimisées localement
continuer $\leftarrow$ *vrai*
tant que continuer **faire**
  Soit gv, bv et av des listes vides
  pour $i \in [1, frac|V|2$ **faire**
    trouver $a \in A$ et $b \in B$ qui maximisent $g(a,b)$
    ajouter $g(a,b)$ à gv, a à av et b à bv
    Ne plus prendre a et b en considération
  fin pour
  Trouver k qui maximise la somme g_{max} de $gv_1, ..., gv_k$
  si $g_{max} \geq 0$ **alors**
    Échanger $av_1, ..., av_k$ avec $bv_1, ..., bv_k$
  sinon
    continuer $\leftarrow faux$
  fin si
fin tant que

La complexité de l'algorithme KL est de $O(n^2)$ (n le nombre de sommets). Fiduccia et Mattheyes [33] ont proposé une amélioration qui utilise la structure de données *bucket list* pour classer les différentes valeurs de g. Cette structure de données implémente une

priorité lors des insertions, suppressions et mises à jour de la liste (figure 4.3). Le vecteur B contient des vecteurs pointant sur des sommets ayant le même indice, stockés dans un *bucket* du vecteur A. Les gains les plus grands et les moins grands sont stockés pour un accès plus rapide. Ces structures permettent un temps d'accès, d'insertions et de sélection des éléments en temps constant, passant ainsi la complexité de l'algorithme KL en $O(m)$ (m le nombre d'arêtes).

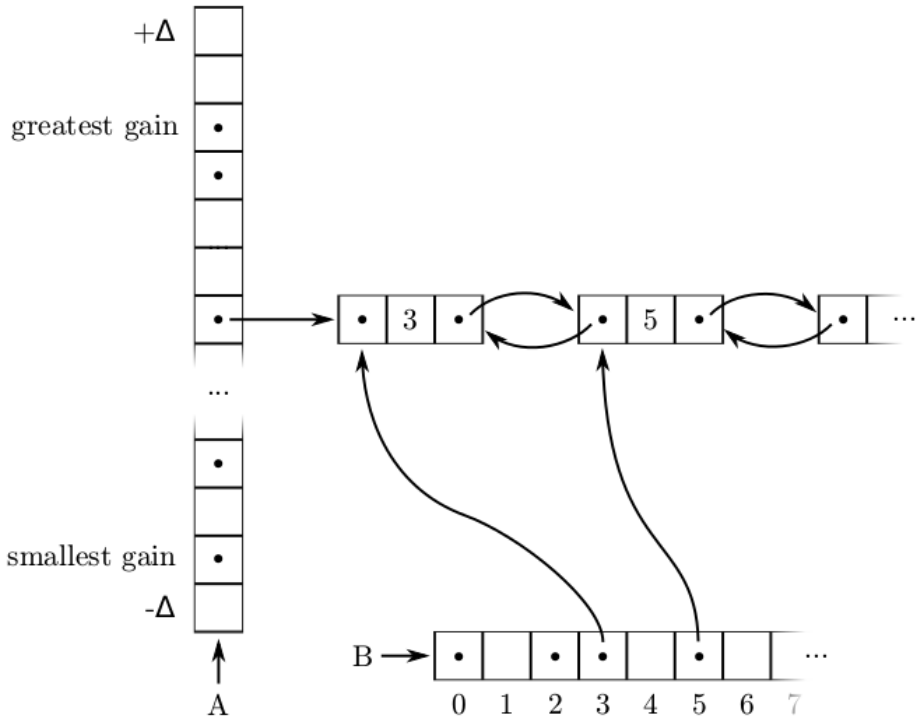

Figure 4.3: La structure de données utilisée par Fiduccia et Mattheyses

L'implémentation proposée par Fiduccia-Mattheyses est efficace et rapide, et a complètement remplacé l'algorithme KL pour l'affinage de bissections. George Karypis et Vipin Kumar [45] proposent une adaptation de l'algorithme au problème du k-partitionnement, qui reprend la notion de gain de l'algorithme KL et la structure de données de l'algorithme FM.

4.2.5 Conclusion

La répartition d'une simulation multi-agent sur plusieurs unités de calcul est un problème complexe. Sa performance dépend de deux facteurs : l'équilibrage de la charge de travail entre les UC d'une part, et la minimisation des coûts de communication entre les UC d'autre part.

Comme nous l'avons vu dans la partie 4.2.2, il existe deux méthodes permettant de distribuer une simulation multi-agent : l'affectation statique des agents à une UC, sans tenir compte de leur position dans l'environnement virtuel, et le découpage et la répartition de l'environnement entre les UC. La première solution a l'avantage de garantir une charge équilibrée, tandis que la deuxième se concentre plus particulièrement sur l'optimisation des communications inter-UC.

La mise en œuvre de la répartition des agents ne pose pas de problème technique particulier. Cependant, dans le cas d'une simulation de trafic, l'environnement dans lequel évoluent les agents est un graphe. Le découpage de graphe est un problème exigeant. C'est pourquoi nous avons effectué un état de l'art afin de présenter les techniques appropriées pour le découpage d'un réseau routier. Nous avons particulièrement débattu de la partition multi-niveaux, qui est à ce jour la technique la plus performante permettant de découper des graphes de taille importante, grâce à une contraction préliminaire du graphe. La figure 4.4 illustre les trois phases (contraction, découpage, expansion) du partitionnement multi-niveaux.

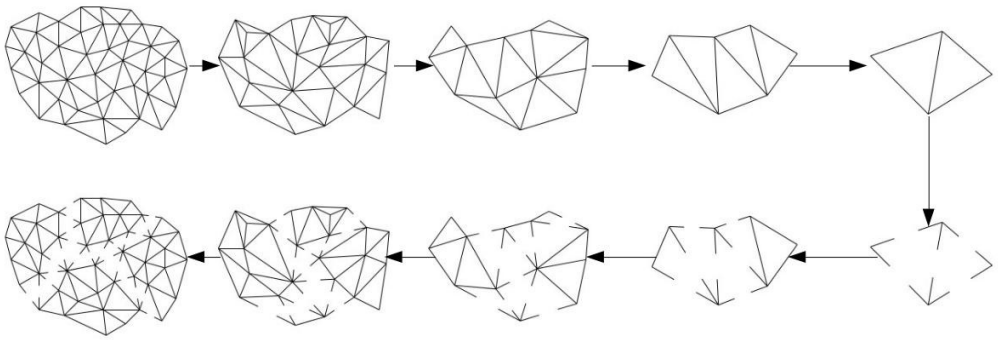

Figure 4.4: Un partitionnement multi-niveaux en action [83]

4.3 Approches proposées

Nous nous intéresserons ici aux moyens à mettre en œuvre pour l'implémentation de deux méthodes de distribution (distribution des agents et découpage de l'environnement) dans un environnement concret. Nous supposerons dans cette partie que nous avons à notre disposition k unités de calcul d'une puissance équivalente.

4.3.1 Distribution orientée agents

La première approche à laquelle nous nous intéresserons sera la division de l'ensemble des agents en k parts égales, de distribuer chaque sous-ensemble sur une unité de calcul et d'exécuter la simulation. Nous avons défini dans le chapitre 3 deux modes de fonctionnement pour notre simulateur de test : macroscopique et microscopique. Chacun d'eux implémente son propre comportement des agents. Ainsi le détail de la distribution différera légèrement pour ces deux modes.

4.3.1.1 Simulation macroscopique

Dans la simulation macroscopique, le temps de parcours sur un arc dépend du nombre d'agents qui le traversent. Les agents ont ainsi besoin de connaître ce nombre à chaque étape pour calculer leur plus court chemin. Dans la distribution par agents, les UC disposent de l'intégralité du réseau routier, mais l'information le concernant est incomplète. En effet, si un agents géré par une autre UC quitte ou arrive sur un arc, il n'en aura a priori pas connaissance.

C'est pourquoi il est nécessaire pour les UC d'échanger cette information. Pour un pas de temps donné, soit in_e le nombre d'agent arrivés sur l'arc e et out_e le nombre d'agents quittant cette arc. Une UC conserve pour chaque arc e l'information suivante :

$$diff_e = in_e - out_e \tag{4.15}$$

À chaque pas de temps, pour tous les arcs e du réseau, chaque UC va communiquer $diff_e$. Ainsi, à tout moment, l'intégralité des UC connaissent l'état de l'ensemble du réseau. Il s'agit des seules communications nécessaires pour cette approche. En effet, les agents ne se déplacent pas d'une UC à l'autre : ils évoluent pendant tout leur cycle de vie au sein de la même UC. En revanche, leur avancement sur le réseau dépend des mouvements des autres agents, gérés par d'autres UC. D'où la communication continue entre les UC pour informer de la dynamique locale de chacun d'eux. Soit I représentant la taille du nombre entier positif nécessaire pour encoder le différentiel de chaque arc. Le coût de communication total à chaque pas de temps, sera de :

$$k.|E|.I \tag{4.16}$$

4.3.1.2 Simulation microscopique

Dans notre modèle microscopique, les agents vont adapter leur vitesse en fonction de l'état de l'agent qui les précède. Cet agent a de grandes chances de se trouver sur une autre UC. L'agent suiveur va alors devoir émettre un requête d'état vers l'UC sur lequel se trouve

l'agent suivi. Pour ce faire, on crée au début de la simulation un index, répertoriant sur quel UC se trouvent chacun des agents. Chaque couple agent suiveur, agent suivi se trouvant sur des unités différents va alors générer une communication inter-UC par pas de temps. Pour adapter sa vitesse, un agent a besoin de trois informations concernant l'agent le précédant : sa position, sa vitesse et s'il est toujours présent sur l'arc (i.e. son arc actuel). Cette information est codée par trois entiers positifs. Le coût de communication lié aux interactions entre les agents est donc de $3.|A|.I$

Lorsqu'un agent arrive sur un nouvel arc, son prédécesseur sera le dernier agent à être arrivé sur cet arc avant lui. C'est pourquoi il sera nécessaire pour les UC de savoir pour chaque arc le dernier agent à être arrivé. Cette mise à jour devra être effectuée à chaque pas de temps. Cette information peut être codée par un entier désignant l'agent en question. Le coût de communication engendré sera donc de $k.|E|.I$.

Enfin, pour le calcul des plus courts chemins, les agents ne vont plus pouvoir compter sur le diagramme fondamental du trafic. On prendra comme valeur de distance le temps mis par le dernier agent à avoir traversé l'arc (comme expliqué dans la partie 3.4.4). Il faut donc également communiquer ces temps de traversée. Le coût sera de $k.|E|.I$.

Le coût de communication à chaque pas de temps de la distribution par agent de la simulation microscopique est donc de :

$$(3.|A| + 2.k.|E|) \times I \tag{4.17}$$

4.3.2 Distribution orientée environnement

Dans le second modèle de distribution, on va découper le réseau routier, puis répartir les parties obtenues sur les UC (figure 4.5). L'avantage de ce modèle est de conserver sur la même unité de calcul les agents qui sont géographiquement proches sur le réseau. Au lieu de distribuer les agents, nous distribuons donc les sommets et les arcs sortants (et donc les agents situés sur ces sommets et ces arcs) de sorte que les agents qui sont situés au même endroit soient sur la même UC. Pour éviter que des agents présents sur la même portion de route se trouvent sur des UC différentes, nous ne couperons jamais les arcs au milieu. La partition est réalisée au début de la simulation, et reste pour le moment inchangée jusqu'à ce qu'elle se termine.

4.3.2.1 Adaptation de l'algorithme *Differential Greedy*

Nous souhaitons obtenir une partition dont chaque sous-partie contient un nombre d'agents équilibré. Pour obtenir cette partition, nous utiliserons l'algorithme DG que nous avons présenté en partie 4.2.4.2. Nous avons modifié l'algorithme (voir algorithme 6), car il n'est pas prévu pour fonctionner avec des sommets pondérés.

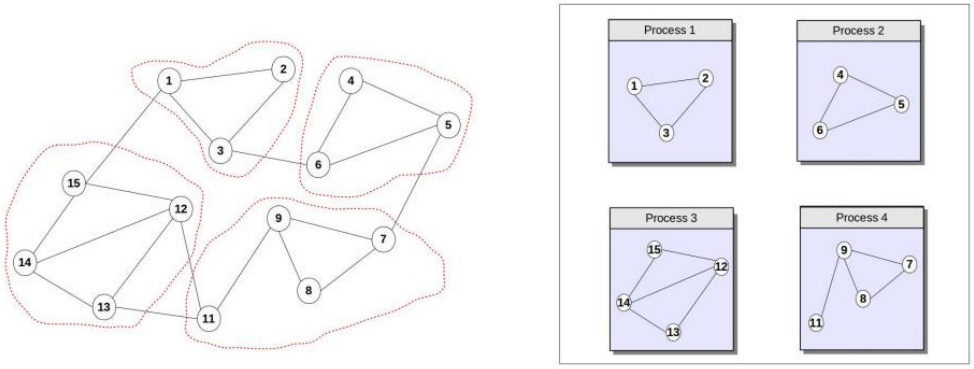

Figure 4.5: Le graphe est partitionné et chacune des parties est distribuée entre les processus disponibles

À chaque itération de la boucle principale, l'algorithme original changeait cycliquement la partition à laquelle il allait ajouter un sommet. Dans notre cas, nous choisirons systématiquement la partition ayant actuellement le moins d'agents.

En outre, l'algorithme DG, de part son critère de choix différentiel, a tendance à produire des sous-parties non connexes. Nous avons ajouté un facteur α pour donner plus d'importance aux poids intérieurs, et ainsi produire des partitions plus connectées.

4.3.2.2 Simulation macroscopique

La distribution de l'environnement de la simulation implique que chaque unité de calcul connaisse uniquement ce qui se passe sur la partie du graphe qu'il gère. Cependant, il est nécessaire pour chaque UC d'avoir une connaissance globale du réseau pour appliquer l'algorithme de Dijsktra. Ainsi, à chaque pas de temps, avant que les agents n'agissent, toutes les UC doivent échanger les informations concernant le nombre d'agents présents sur chacun des arcs qu'elles gèrent. Le coût des communications du poids des arcs est donné par la relation : $|E|.I$.

De plus, lorsqu'un agent se rend sur une partie du réseau qui n'est pas gérée par l'UC sur laquelle il se trouve, il doit être transféré sur l'UC correspondante. Un agent peut être codé avec trois entiers (son identifiant, son emplacement actuel et sa destination). Soit n le nombre de migrations pour un pas de temps. Ainsi, le coût de la migration des agents est de $3.I.n$. Il y a en moyenne $|A|/|E|$ agents par sommet. Soit E_c l'ensemble des arcs dont les sommets sont sur deux hôtes différents. Nous avons en moyenne $n = \frac{|A|}{|E|}|E_c|$.

On peut ainsi estimer le coût de communication de la distribution de l'environnement

Algorithm 6 Algorithme *Differential Greedy* modifié

Pré-conditions : Graphe $G = (V, E)$, taille k de la partition
Post-conditions : Partition P
$P \leftarrow P_0, ..., P_{k-1}$
$V' \leftarrow V$
pour $p \in [0, k-1]$ **faire**
  $v \leftarrow$ un sommet aléatoire de V'
  $P_p \leftarrow \{v\}$
  $V' \leftarrow V' \setminus \{v\}$
fin pour
tant que $|V'| > 0$ **faire**
  $p \leftarrow$ indice de la partition ayant le moins d'agents
  $m = min_{v \in V'} \alpha.int(v, p) - ext(v, p)$
  $S = \{v \in V'\} | \alpha.int(v, p) - (ext(v, p) = m$
  $mv =$ un sommet aléatoire de S
  $P_p \leftarrow P_p \cup \{mv\}$
  $V' \leftarrow V' \setminus \{mv\}$
fin tant que
Retourner P

appliquée à la simulation macroscopique à :

$$(3\frac{|A|}{|E|}|E_c| + |E|) \times I \tag{4.18}$$

4.3.2.3 Simulation microscopique

La modèle microscopique requiert des communications très similaires au modèle macroscopique lorsque l'on utilise le découpage de l'environnement. Le coût de transfert des agents est le même, c'est à dire $3\frac{|A|}{|E|}.|E_c|.I$. Pour le calcul des chemins, les agents ont besoin des informations concernant le temps de parcours du dernier agent à avoir traversé chaque arc. Les UC doivent communiquer cette information, et le coup s'élève à $|E|.I$

On a donc un coût total identique à celui de la simulation microscopique :

$$(3\frac{|A|}{|E|}|E_c| + |E|) \times I \tag{4.19}$$

4.3.3 Conclusion

Nous avons présenté deux méthodes permettant de distribuer notre simulateur. Pour chacune des méthodes, nous avons évalué les communications inter-UC qui seront nécessaires pour les deux modes de fonctionnement du simulateur (macro/micro). Le tableau 4.1 donne un récapitulatif de ces communications.

	Microscopique	Macroscopique
Environnement	Transfert des agents Mise à jour du réseau	Transfert des agents Mise à jour du réseau
Agents	Requêtes états agents Mise à jour du réseau Mise à jour des derniers arrivants	Mise à jour du réseau

Tableau 4.1: Récapitulatif des communications nécessaires pour les différentes combinaisons

4.4 Implémentation des modèles de distribution

Afin de tester les solutions développées, nous les avons implémentées puis déployées sur un cluster de calcul expérimental, afin de simuler un passage à l'échelle sur des systèmes à mémoire distribuée plus performants (grâce notamment aux infrastructures fournies par le cloud computing). Nous avons appliqué les modèles de distribution que nous avons présentés dans la section précédente sur notre simulateur.

Pour les communications inter-processus, nous utilisons MPI (*Message Passing Interface*), qui est le standard *de facto* pour le calcul parallèle sur architecture distribuée. MPI offre un modèle de communication standardisé entre les différents processus d'un programme et a de nombreuses implémentations efficaces qui s'exécutent sur une grande variété de machines. Notre simulateur étant développé en python, nous utilisons MPI4PY, qui est une interface efficace et éprouvée qui permet d'utiliser MPI avec Python.

Une fois les paramètres de la simulation et le réseau routier chargés, la classe *Simulation* va initialiser MPI et créer un processus par unité de calcul disponible. Les éléments de la simulation seront partagés entre les processus en fonction de la méthode de distribution choisie. C'est ensuite la classe *Context* qui va gérer les communications entre les processus à chaque pas de temps (figure 4.6).

4.4.1 Les primitives de communications MPI

Comme nous l'avons vu dans la section 2.4.2, MPI fournit un ensemble de procédures standards destinées à la communication par passage de messages. Nous détaillons ici les primitives MPI que nous avons utilisées pour implémenter les communications nécessaires à chacun des modes de fonctionnement du simulateur (micro et macro).

4.4.1.1 Envoi réception point à point

Le mode de communication basique de MPI est l'envoi/réception de messages d'un processus à un autre. Toutes les autres procédures peuvent être implémentées grâce à

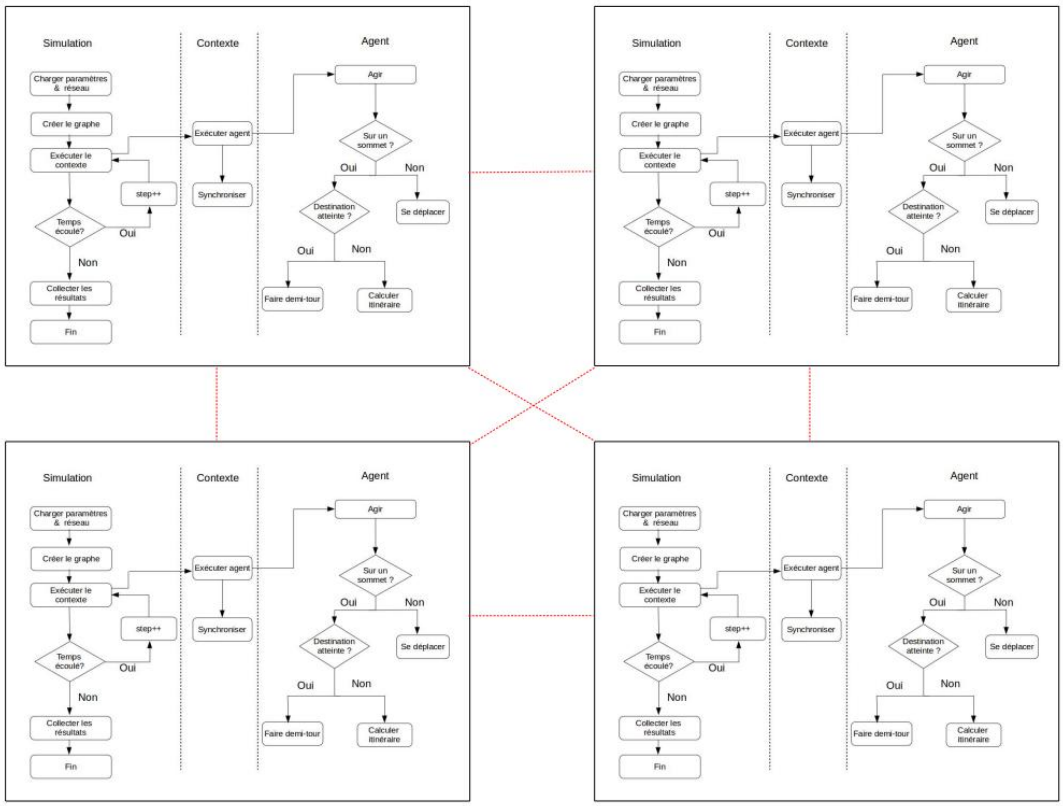

Figure 4.6: Distribution de la simulation sur plusieurs UC

l'envoi/réception point à point. Lorsqu'un processus A décide d'envoyer un message à un processus B, il utilise la fonction $MPI\_Send()$, qui précise les données à envoyer et le rang du processus auquel ces données sont envoyées. Il arrête son exécution jusqu'à ce que les données soient transmises. À son tour, le processus B doit indiquer qu'il est prêt à recevoir à l'aide de la fonction $MPI\_Receive()$. Il arrête lui aussi son exécution jusqu'à ce que les données soient reçues. Lorsque c'est le cas, il envoie un accusé de réception au processus A, et les deux processus reprennent leurs exécutions normales.

Ce mode d'envoi/réception, où les processus attendent tous deux que le message soit transmis est le mode par défaut de MPI. Ces communications sont dites bloquantes. Il est également possible d'utiliser des communications non bloquantes, lorsque l'échange des données n'est pas critique pour la poursuite de l'exécution du code. C'est le mode non bloquant, que l'on utilise en invoquant les fonctions $MPI\_ISend()$ et $MPI\_IReceive()$.

4.4.1.2 Communications collectives et synchronisations

Nous venons de décrire les communications point à point impliquant deux processus. Bien qu'elles soient au cœur du fonctionnement de MPI, il est souvent utile de définir des communications impliquant la participation de tous les processus en même temps. Ceci nécessite des processus de synchronisation, permettant de s'assurer que tous les processus ont atteint un certain point du code avant de poursuivre l'exécution. Les processus ayant atteint ce point en premier vont attendre tous les autres (figure 4.7).

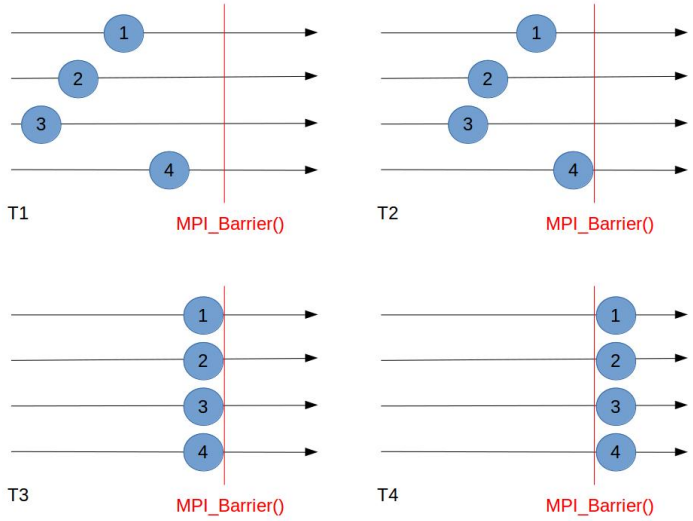

Figure 4.7: Synchronisation MPI à l'aide de $MPI\_Barrier()$

4.4.1.3 Diffusion

Un scénario de communication collective communément rencontré est lorsqu'un processus doit diffuser la même information à l'ensemble de ses semblables (figure 4.8). MPI implémente ce comportement avec la fonction $MPI\_Bcast()$. Cette fonction prend en paramètre la donnée à diffuser ainsi que le processus racine. Tous les processus font appel à la même fonction. Si le processus appelant est la racine, il diffuse la donnée. Dans le cas contraire, il la reçoit.

4.4.1.4 Répartition et collecte de données

Dans certains cas, un processus peut vouloir découper une information avant de la communiquer aux autres processus. C'est ce que permet $MPI\_Scatter()$. Cette fonction fonctionne de manière similaire à $MPI\_Bcast()$, à la différence qu'elle prend un tableau

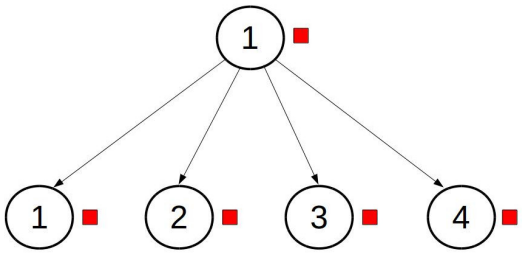

Figure 4.8: Diffusion de message grâce à $MPI\_Bcast()$

de taille k en paramètre, et communique une cellule du tableau à chacun des k processus (figure 4.9).

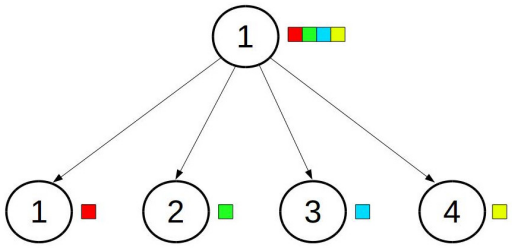

Figure 4.9: Répartition d'information avec $MPI\_Scatter()$

La fonction $MPI\_Gather()$ fournit la fonction inverse, c'est à dire qu'elle permet à un processus de collecter dans un tableau des données provenant des autres processus (figure 4.10). Le découpage de données, leur distribution pour le traitement par des processus séparés, puis la collecte des résultats est une procédure souvent employée dans le calcul distribué [26].

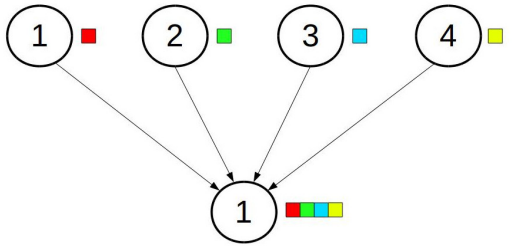

Figure 4.10: Collecte d'information avec $MPI\_Gather()$

Enfin, nous pouvons mentionner $MPI\_Allgather()$. Cette fonction est utilisée lorsque chaque processus possède une partie de l'information et qu'ils ont tous besoin de l'intégralité de l'information (figure 4.11). Les données sont collectées dans un tableau, classées selon le rang de leur processus d'origine, qui sera disponible pour tous les processus.

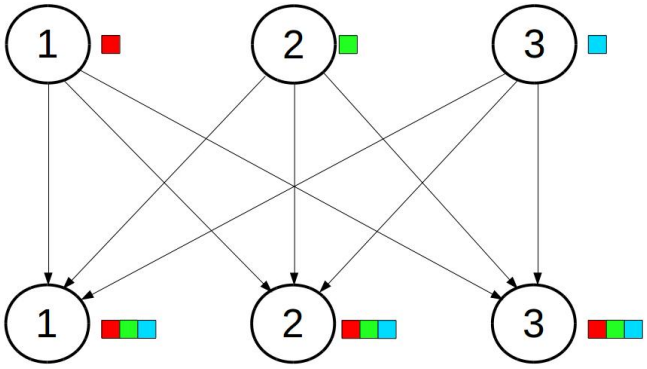

Figure 4.11: Collecte d'information avec $MPI\_Allgather()$

4.4.2 Communications avec la méthodes de distribution des agents

La première chose à faire une fois que le graphe représentant le réseau routier est chargé par le processus principal, c'est de le diffuser à l'ensemble des processus grâce à $MPI\_Bcast()$. Nous créons ensuite les agents et les répartissons dans k sous-ensembles. Ces sous-ensemble sont ensuite répartis avec $MPI\_Scatter()$, de sorte que chaque processus gère un sous-ensemble d'agents. La simulation est ensuite lancée. Les communications seront différentes en fonction du modèle utilisé, ainsi que nous l'avons décrit dans la partie 4.3.

4.4.2.1 Simulation macrocopique

Nous devons à chaque pas de temps, mettre à jour le nombre d'agents qui se trouvent sur les arcs du réseau. Chaque processus possède une partie de cette information (les arrivées et les départs des agents qu'il gère) et a besoin de l'intégralité de cette information. Nous avons ainsi utilisé $MPI\_Allgather()$ qui a spécifiquement été conçue pour ce type de situations.

4.4.2.2 Simulation microcopique

La distribution des agents appliquée à la simulation microscopique nécessite trois types de communication : les temps de parcours mis à jour en fonction du temps mis par les derniers agents arrivés, la mise à jour des derniers agents arrivés sur les arcs, et les requêtes d'état émises par les agents pour connaître la vitesse et la position de l'agent les précédant. Les deux premières communications sont gérées à chaque pas de temps par le contexte grâce à un $MPI\_Allgather()$ réunissant l'information distribuée.

Les requêtes d'état sont plus difficiles à gérer. En effet, à tout moment, un agent situé sur une UC pourra avoir besoin de l'état d'un agent se trouvant sur une autre UC, qui devra alors être en mesure de répondre. C'est pourquoi lorsqu'ils sont créés, les contextes vont lancer un *thread* serveur qui sera destiné spécialement à répondre aux requêtes d'état. Quand un agent situé sur une UC A nécessite de l'information concernant un agent situé sur une UC B, l'UC A va émettre un $MPI\_Send()$ vers B. Le *thread* lancé par le contexte de B va alors intercepter la requête, puis empaqueter l'information demandée et la renvoyer vers A, qui pourra reprendre la simulation. Tout ceci sans impacter le déroulement de la simulation sur B.

Afin de s'assurer de la cohérence de la simulation, nous avons aussi placé des barrières à chaque pas de temps. On empêche ainsi qu'un agent d'une UC au temps de simulation t ne reçoive une information d'état d'un agent se trouvant à une UC au temps $t + 1$.

4.4.3 Communications avec la méthode du découpage de l'environnement

Après le chargement du réseau routier par le processus principal, les agents sont créés et positionnés sur le réseau. Celui-ci est découpé en k partie de poids équilibré grâce à l'algorithme présenté en partie 4.3.2.1. La structure du réseau est ensuite diffusée ($MPI\_Bcast()$) à toutes les unités (les agents auront besoin de cette structure pour appliquer l'algorithme de Dijsktra). On diffuse alors la partition obtenue, de sorte que chaque UC connaisse la partie du graphe qu'elle doit gérer, et quelle partie est gérée par quelle unité. Les agents sont ensuite répartis ($MPI\_Scatter()$) sur les UC en fonction de leur localisation. Quelque soit le modèle utilisé, les communications restantes sont assez similaires. Elles se divisent en deux catégories : mise à jour du réseau et transfert des agents.

Les agents présents sur une UC ont besoin de connaître l'état de l'ensemble du graphe afin de calculer leurs chemins, que l'on soit dans le modèle micro (derniers temps de parcours sur les arcs) ou macro (nombres d'agents sur les arcs). Cette étape de synchronisation est gérée à chaque pas de temps par le contexte, en utilisant $MPI\_Allgather()$, de la même manière que dans la méthode de distribution des agents.

Le type de communication spécifique à la distribution de l'environnement est le transfert des agents. Ces derniers vont en effet être déplacés au fur et à mesure de leur évolution

sur le réseau de transport. Lorsqu'ils se déplacent sur une partie du graphe qui n'est plus gérée par leur UC actuelle, ils doivent être transférés sur l'UC correspondante. Les UC gardent en mémoire les agents à transférer, et les transferts effectifs sont tous réalisés en même temps, à la fin du pas de temps courant. Chaque agent est envoyé à l'UC qui lui correspond de manière non bloquante ($MPI\_ISend()$). L'UC attend ensuite les agents envoyés par les autres UC ($MPI\_Receive()$). Les agents envoyés sont retirés du contexte, et les agents reçus y sont ajoutés.

4.4.4 Déploiement du simulateur sur un cluster

Une fois les communications par passage de message implémentées, notre simulateur est prêt à être déployé sur un système distribué. Nous n'avions malheureusement pas accès à un grand nombre de serveurs, ce qui aurait été l'infrastructure idéale pour expérimenter sur un système MIMD.

Nous avons mis en place un cluster de test, en utilisant deux serveurs sous Linux Mint 17.2 Rafaela (noyau 3.16.0-38-generic) chacun doté de deux processeurs Intel Xeon E7-4820 (16 cœur à 2 GHz), et disposant de 250 Go de mémoire vive. Ces serveurs sont sur le même switch d'un réseau ethernet. Nous disposions ainsi de 64 cœurs de calcul.

Un serveur ssh est installé sur chacune de ces machines. Le protocole ssh sera utilisé par MPICH 3.0.4, pour encapsuler les messages MPI. Le code de la simulation, ainsi que le réseau utilisé sont hébergés sur un serveur NFS externe, que nous avons également installé.

4.5 Expérimentations

Afin de mesurer l'efficacité de nos méthodes de distribution, nous avons comparé les temps d'exécution obtenus avec les temps d'exécution que l'on obtient dans les mêmes conditions en séquentiel (avec un seul processeur).

4.5.1 Conditions des tests réalisés

Pour nos tests, nous avons lancé une simulation de 100 pas de temps sur un réseau routier représentant la zone Paris-Saclay. Nous avions tout d'abord utilisé des réseaux invariant d'échelle générés aléatoirement. Cependant la topologie du réseau utilisé est cruciale pour la distribution de la simulation. C'est pourquoi nous avons décidé d'utiliser un réseau urbain réel, afin que nos résultats soient le plus représentatif possible.

Le réseau routier que nous utilisons pour nos simulation représente une zone englobant Saclay, Versailles et Satory (figure 4.12), habitée par 650000 personnes. Il s'agit d'un graphe orienté connexe, comportant 3784 arcs et 1856 sommets. Un itinéraire avec une

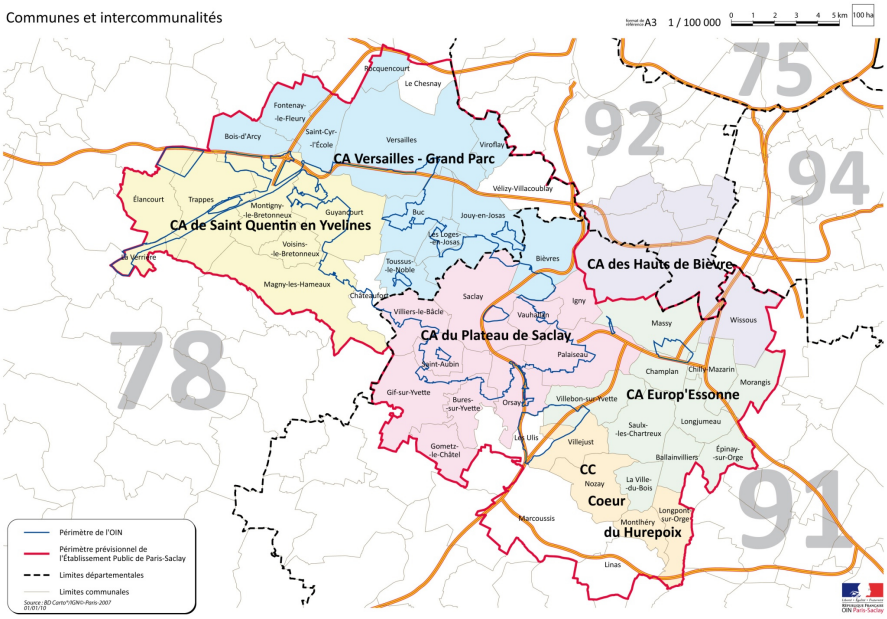

Figure 4.12: Zone Paris-Saclay (http://www.u-psud.fr)

origine et une destination choisies aléatoirement dans le réseau est en moyenne de 12.3km (distance calculée en faisant la moyenne des distances de 1000000 d'itinéraires aléatoires). La figure 4.13 montre la structure du réseau.

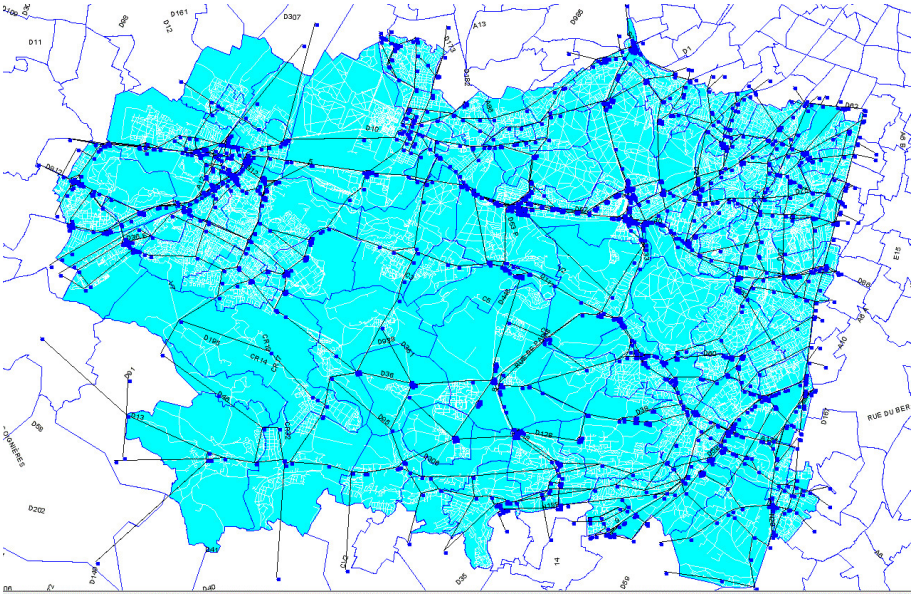

Figure 4.13: Structure du réseau Paris-Saclay

Pour mesurer le passage à l'échelle de nos méthodes de distribution, nous les avons ap-

pliquées sur les modèles micro et macro avec un nombre croissant d'agents. Le nombre d'usagers du réseau Paris-Saclay est environ de 110.000. Nous considérons dans nos tests de 10.000 à 500.000 agents, ce qui représente entre 10% et 500% du volumes réel d'usagers du réseau.

4.5.2 Résultats et interprétation

Nous avons exécuté des simulations impliquant un nombre croissant d'agents sur les modèles macroscopique et microscopique. Nous faisons figurer dans le tableau 4.2 les résultats obtenus pour chacun des modèles pour une simulation en séquentielle (1 cœur de calcul) ainsi qu'avec les méthodes de distribution présentées dans ce chapitre (sur 64 cœurs de calcul). Chaque simulation a été exécutée dix fois, afin de minimiser les effets de l'aléatoire dans nos tests. Les chiffres du tableau sont les moyennes des temps d'exécution (en secondes) pour les dix instances réalisées. L'écart type est à chaque fois inférieur à 5% de la moyenne des temps.

Nombre d'agents	10 000	50 000	100 000	250 000	500 000
Macro sequentiel (1 cœur)	30,9	142,8	288,3	714,3	1540,9
Distribution agents Macro (64 cœurs)	8,9	15,5	26,4	57,4	109,4
Distribution environnement Macro (64 cœurs)	11,3	25,2	46,5	108,2	200,5
Micro sequentiel (1 cœurs)	62,6	302,4	642,3	1686,2	3413,4
Distribution agents Micro (64 cœurs)	76,3	348,8	690,7	1747,9	3434,7
Distribution environnement Micro (64 cœurs)	15,8	57,1	109,8	298,7	574,3

Tableau 4.2: Temps d'exécution (en secondes) d'une simulation de 100 pas de temps sur le réseau de Paris-Saclay

Les accélérations de temps d'exécution mesurées entre l'exécution séquentielle et les deux méthodes de distribution sont indiquées sur la figure 4.14 pour la distribution des agents et sur la figure 4.15 pour la distribution de l'environnement. L'accélération mesure combien de fois la simulation est plus rapide entre la méthode de distribution considérée et une implémentation séquentielle. L'accélération pour k processeurs est donnée par :

$$S_k = \frac{t_1}{t_k} \tag{4.20}$$

Ainsi, nous pouvons par exemple lire sur la figure 4.14 que pour 200.000 agents, la simulation macroscopique est environ 12 fois plus rapide lorsque l'on distribue les agents que lorsque la même simulation est exécutée de manière séquentielle.

On peut constater que, malgré le peu de matériel mis en œuvre pour nos expérimentations, ces deux méthodes améliorent significativement le temps d'exécution de notre simulateur de mobilité multi-agent dans sa version macroscopique. On obtient en effet avec 500000 agents une accélération de 8 avec la distribution de l'environnement, et de 14 avec la distribution des agents. Celle-ci est particulièrement adaptée à la simulation macroscopique.

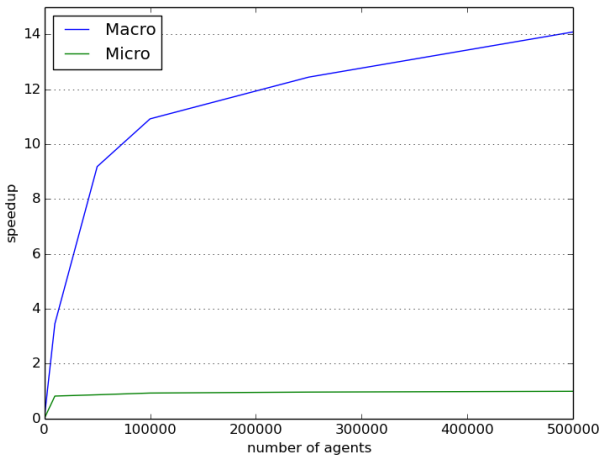

Figure 4.14: Accélération pour la distribution des agents

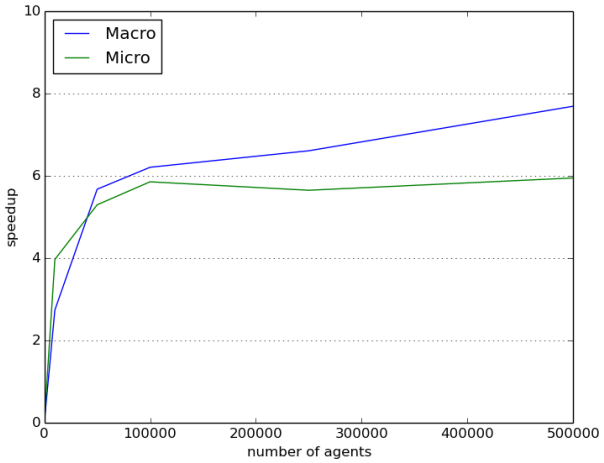

Figure 4.15: Accélération pour la distribution de l'environnement

En effet, cette méthode fournit une répartition idéale de la charge de travail. De plus, dans le modèle macroscopique, les agents n'interagissent pas directement entre eux. Il n'y a donc aucun inconvénient à répartir arbitrairement les agents sur les processeurs à disposition.

En revanche, la distribution des agents n'est plus du tout efficace dans le contexte d'une simulation microscopique. Cela s'explique par le grand nombre d'interactions locales entre les agents dans ce modèle. Si deux agents amenés à interagir se trouvent sur des processeurs différents, cela va engendrer de nombreuses communications. Le coût de ces communications va outrepasser le bénéfice engendré par la distribution du travail sur de

nombreux processeurs, et cela va parfois même se révéler contre-productif ($S_k < 1$).

C'est pourquoi dans le cas d'une simulation microscopique, il est recommandé de s'efforcer de conserver les agents situés dans des endroits proches dans la simulation sur la même UC. C'est ce que fait le distribution de l'environnement. Cette méthode va limiter les communications, ce qui rend la distribution à nouveau bénéfique ($S_{64} = 6$ pour 500000 agents).

4.6 Conclusion

Dans ce chapitre, nous avons présenté deux méthodes permettant de partager l'exécution d'un simulateur de trafic multi-agent sur un système à mémoire distribuée. La première méthode consiste à répartir les agents arbitrairement sur les unités de calcul disponibles (figure 4.16), sans tenir compte de la localisation des agents. Avec cette méthode, chaque UC a une copie de l'intégralité de l'environnement.

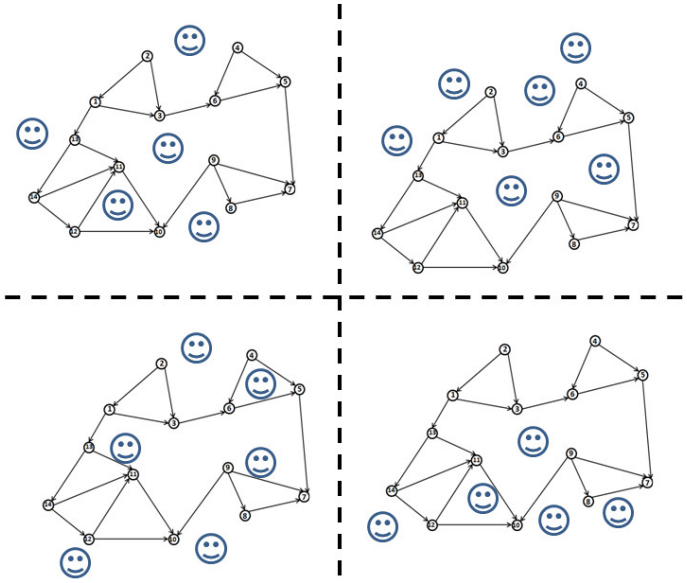

Figure 4.16: Répartition des agents

La seconde va découper et distribuer l'environnement de la simulation sur ces UC (figure 4.17). Cette méthode s'efforce de garder les agents proches géographiquement sur la même UC. Chaque UC a cette fois-ci accès uniquement à une portion de l'environnement.

Nous avons implémenté et appliqué ces méthodes de distribution à deux modèles de simulateur : macroscopique et microscopique. Le premier représente les simulateurs se fondant sur un diagramme fondamental pour déduire les vitesses des véhicules, tandis que le deuxième représente les simulateurs utilisant un modèle de poursuite. Ces deux modèles

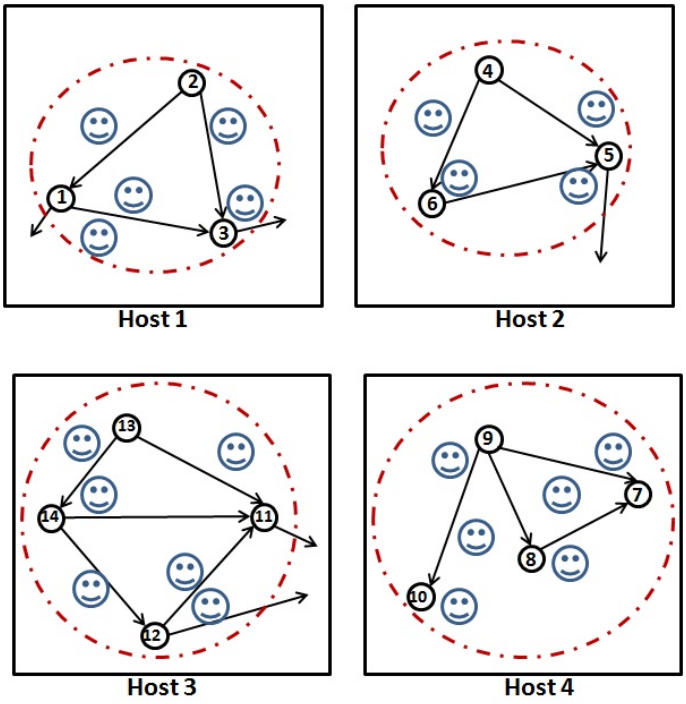

Figure 4.17: Répartition de l'environnement

sont génériques et représentatifs de la grande majorité des simulateurs de mobilité existant dans la littérature.

Les résultats obtenus sont très bons pour la répartition des agents sur un simulateur macroscopique. Nous avons en effet un scénario idéal, avec une charge de travail parfaitement répartie et très peu de communication inter-UC. Cependant, cette méthode de distribution n'est plus efficace pour le grand nombre de simulateurs nécessitant de nombreuses interactions locales entre les agents. Il convient dans ce cas là d'utiliser la méthode de distribution de l'environnement.

La distribution de l'environnement est actuellement réalisée statiquement, au début de la simulation. Les agents seront amenés à se déplacer entre les UC au cours de la simulation. Si de nombreux agents sont concentrés sur la même UC, cela engendre un déséquilibre de charge entre les UC, ce qui sera nuisible aux performances globales de la simulation. Le chapitre suivant sera consacré à la mise en place d'un mécanisme d'équilibrage de charge dynamique, permettant d'améliorer encore les performances de la distribution de l'environnement appliquée à la simulation microscopique.

Chapitre 5

Diffusion dynamique de charge

Sommaire

5.1 Introduction

Le chapitre précédent propose des méthodes permettant de distribuer l'exécution d'un simulateur de trafic multi-agent sur de nombreuses unités de calcul. L'une de ces méthodes est la distribution des agents de manière arbitraire sur les UC. Cette méthode produit une distribution où la charge de travail, représentée par le nombre d'agents présents sur chaque serveur, est parfaitement équilibrée entre les UC tout au long de la simulation. Elle fonctionne particulièrement bien dans le cas d'une simulation où les agents ne sont pas amenés à communiquer directement.

Cependant, dans le cas d'une simulation microscopique, utilisant un modèle de poursuite, la distribution des agents n'est pas envisageable. Elle génère un nombre important de

communications inter-UC, ce qui nuit aux performances du système distribué. Il convient pour ce type de simulations de découper l'environnement en sous-parties que l'on distribuera entre les unités de calcul.

Cette méthode fournit des résultats acceptables, mais comporte un inconvénient : les agents, qui sont mobiles dans leur environnement, vont se trouver au cours de la simulation sur différentes parties du graphe représentant le réseau de trafic. Il vont ainsi se déplacer d'UC en UC. Une simulation de trafic est susceptible de générer ponctuellement des concentrations importantes d'usagers sur certains points, lorsque, par exemple, les utilisateurs du réseau se rendent massivement le matin des quartiers résidentiels aux quartiers administratifs, et inversement le soir. Si une UC est en charge d'une partie de l'environnement qui est beaucoup sollicitée à un moment donné, elle aura alors une charge de travail plus importante que ses semblables.

Une telle situation nuit aux performances globales du système. En effet, chaque sous-partie de la simulation a besoin d'information sur l'état des autres sous-parties pour pouvoir garder un état cohérent. À chaque pas de temps, les sous parties doivent se synchroniser en échangeant des informations, selon les mécanismes décrits dans le chapitre 4. Ce processus de synchronisation nécessite d'attendre que chaque processus ait fini l'exécution de son pas de temps courant. Ainsi le processus le plus rapide devra impérativement attendre à chaque pas de temps le processus le plus lent. La figure 5.1 illustre ce phénomène pour deux processus.

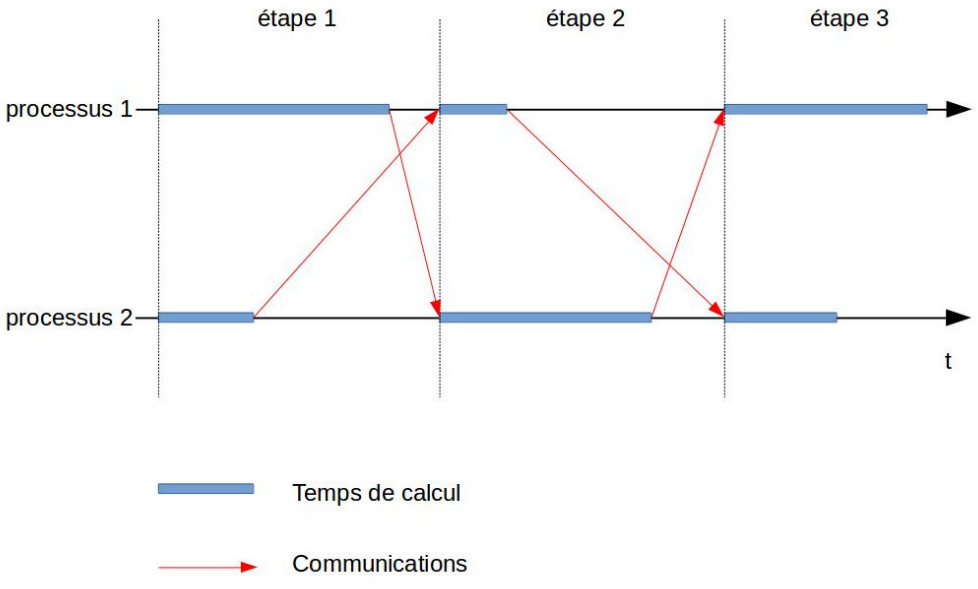

Figure 5.1: Synchronisation du simulateur

L'état du réseau évoluant dynamiquement au cours de la simulation, il n'est pas possible d'obtenir une simulation de charge équilibrée en découpant son environnement de manière définitive lorsqu'elle débute. Ce chapitre traitera des solutions que nous avons mises en œuvre pour palier ce problème.

5.2 Équilibrage dynamique de charge

Pour garantir des performances optimales de la distribution de la simulation de trafic microscopique, nous devons mettre en place un mécanisme permettant de redistribuer la charge de manière dynamique (pendant la simulation).

Les algorithmes d'équilibrage de charge peuvent être classés en deux catégories : centralisés et décentralisés [52]. Dans les algorithmes centralisés, une entité centrale collecte les informations de charge de toutes les UC, redécoupe le problème, et redistribue la charge aux UC. Les algorithmes sont dits décentralisés lorsque la charge est transférée de point à point, selon des décisions résultant de communications locales entre les UC.

De manière générale, le processus d'équilibrage de charge dynamique se déroulent en plusieurs phases [85], représentant chacune une des trois décisions à prendre pour l'équilibrage de charge dynamique :

- Initiation : détermine quand déclencher le processus d'équilibrage de charge.
- Emplacement : décide quelles sont les UC qui échangeront de la charge de travail.
- Sélection : sélectionne les tâches à transférer.

Nous avons modélisé notre problème sous la forme d'un graphe. Cela signifie que la phase de sélection consistera à choisir les sommets (et les arcs correspondants) qui devront être transférés. Nous devrons donc à chaque fois qu'un processus d'équilibrage est initié, redécouper le graphe et réaffecter chacune de ses parties aux unités de calcul.

5.3 Le repartitionnement de graphe

Comme pour le découpage de graphe initial, la charge de travail devra être répartie équitablement entre les UC, et les communications inter-UC futures devront être minimisées. Nous avons cependant deux nouvelles contraintes qui sont spécifiques au redécoupage dynamique : le calcul de la nouvelle partition doit être le plus rapide possible, et le temps nécessaire aux transferts des données doit être minimal. Ces deux contraintes sont là pour s'assurer que le processus d'équilibrage ne ralentisse pas la simulation.

En d'autres termes, nous devons minimiser la quantité suivante [77] :

$$T_{total} = \alpha(T_{calcul} + T_{comm}) + T_{repart} + T_{mig} \tag{5.1}$$

Avec :

α : nombre d'itérations avant qu'un nouveau partitionnement ne soit nécessaire;

T_{calcul} le temps d'une d'exécution d'un pas de temps sur l'UC la plus lente;

T_{comm} le temps des communications inter-UC;

T_{repart} le temps de calcul de la nouvelle partition ;

T_{mig} le temps de transfert des données lors du repartitionnement.

Le processus de repartitionnement n'est en général pas effectué après un nombre fixe d'itérations. Il est plutôt déclenché lorsqu'un déséquilibre trop important est constaté, soit par le processus surchargé (*sender-initiated*), soit par un processus n'ayant pas assez de travail (*receiver-initiated*) [30]. Le facteur α représente alors le nombre moyen d'itérations entre chaque déclenchement du repartitionnement.

Si ce facteur α est élevé, cela signifie que la simulation met du temps à se déséquilibrer, et donc que le processus de repartitionnement se déclenche rarement. Dans ce cas, il convient de privilégier la qualité du partitionnement, quitte à passer plus de temps à le mettre en place ($T_{repart} + T_{migr}$). Au contraire, pour les simulations très dynamiques, qui se déséquilibrent rapidement et nécessitent de nombreuses repartitions, il est plus avisé de minimiser le temps nécessaire au repartitionnement, même si l'on doit pour cela sacrifier en qualité de partition. Celle-ci se dégradera dans tous les cas très rapidement.

5.3.1 Le *Scratch-Remap*

À chaque fois que le processus d'équilibrage de charge est déclenché, la méthode intuitive pour rééquilibrer la charge est de repartitionner le graphe de zéro, en utilisant les techniques présentées dans la section 4.2.3. Cette méthode produit en effet une partition d'excellente qualité, que ça soit au niveau du poids de coupe ou de la répartition équitable de la charge. Elle comporte en revanche dans cette forme un inconvénient : le redécoupage est fait sans se soucier de la position actuelle des sommets. Le coût de migration des sommets n'est ainsi pas pris en compte.

C'est pourquoi des techniques de *remapping* intelligents ont été proposées [71]. Le graphe déséquilibré est repartitionné de zéro, puis les partitions sont remappées sur les différents processeurs afin de limiter les transferts de données. Cette méthode est appelée *scratch-remap*. Ces techniques ont été améliorées afin d'obtenir un volume de migration encore

plus faible en affinant le grain de la partition [14], ou en contraignant le partitionnement multi-niveaux [78].

La figure 5.2 montre un exemple de *remapping* de partition. La partition (a) est déséquilibrée, car sa partie 2 comporte 4 sommets, alors que la partie 1 en comporte 3, et la partie 3 seulement 2. Le graphe est donc repartitionné de zéro (b). On n'a pas tenu compte de la partition initiale, donc un grand nombre de migration de sommets seraient nécessaires. Dans (c), on remappe la partition obtenue en échangeant les numéros de partition. Un grand nombre de transferts peut être évité grâce à cette technique.

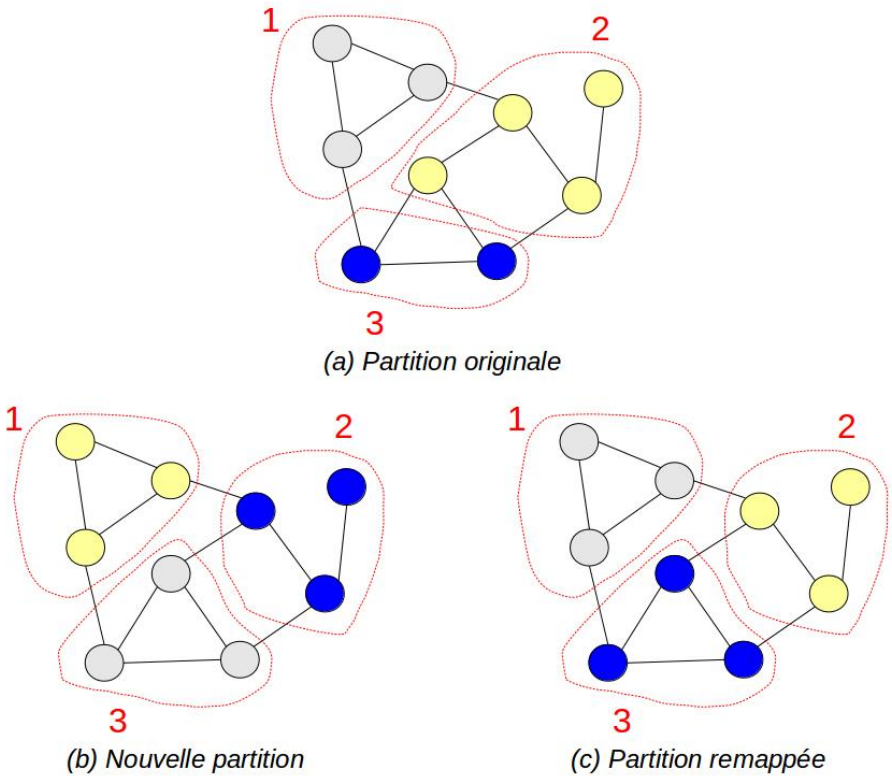

Figure 5.2: Exemple de *Scratch-Remap*

5.3.2 Le repartitionnement diffusif

Le repartitionnement de zéro fournit une nouvelle partition de très bonne qualité. Cette solution est ainsi très utilisée dans les problèmes qui ne nécessitent pas de redécoupage fréquent. Cependant, même dans leurs versions les plus optimisées, les algorithmes de *scratch-remap* entraînent des temps d'exécution importants pour repartitionner le graphe. En outre, la nature centralisée de ce type de solutions implique de nombreuses communica-

tions dirigées vers le même nœud (celui qui sera en charge du calcul de partition). Il n'est donc pas envisageable d'utiliser le *scratch-remap* pour des problèmes très dynamiques [27].

Des solutions ont été proposées pour équilibrer la charge de manière complètement décentralisée. Le processus de décisison est présent dans chacune des UC, qui ne dispose par définition que d'une connaissance locale du problème. Pour cela, on modélise le processus d'équilibrage de charge sous la forme du phénomène physique de diffusion. Par exemple, [24] utilise des équations de diffusion de la chaleur pour répartir la charge de proche en proche.

Le principe de la diffusion est d'équilibrer les distributions non-homogènes par le déplacement local d'entités élémentaires dans la direction suggérée par l'objectif de minimisation d'énergie [23]. Autrement dit, lorsqu'une UC est surchargée, elle transfère une partie de sa charge de travail aux UC l'entourant. Un exemple de cette stratégie est illustré par la figure 5.3. La partie 2 est initialement (a) en surcharge, devant gérer quatre sommets. L'un de ses voisins (la partie 3) ne gère que deux sommets à ce moment-là. Après un échange pair à pair d'informations, un sommet est transféré de la partie 2 vers la partie 3 (b) afin d'obtenir une nouvelle partition du graphe (c).

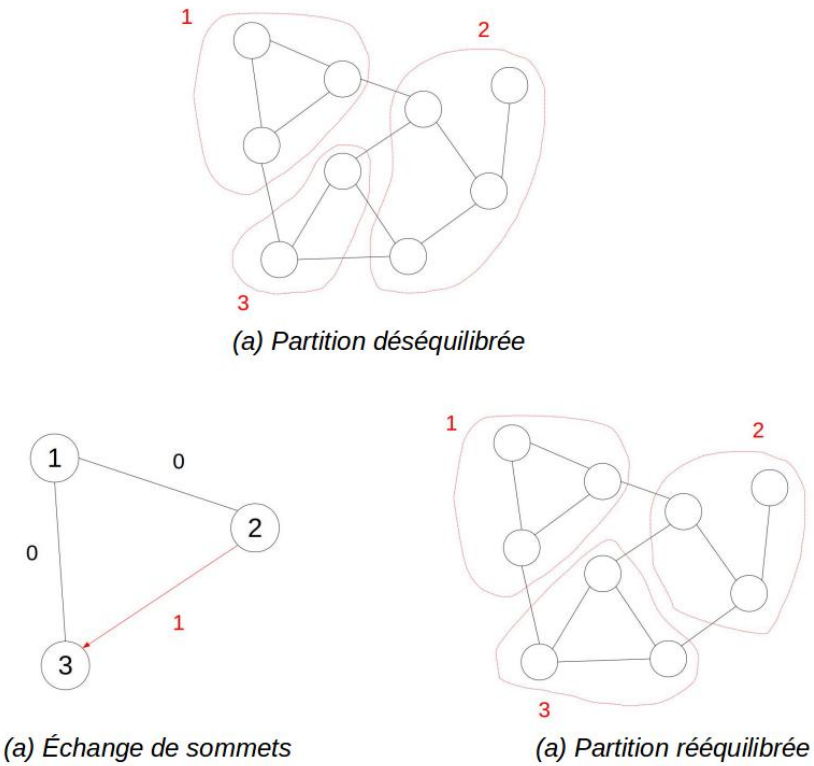

Figure 5.3: Exemple de diffusion de charge

5.4 Équilibrer la charge d'une simulation de trafic microscopique

Dans la simulation de trafic multi-agent distribuée selon l'environnement, présentée dans le chapitre 4, la partition du réseau routier est effectuée une seule fois, basée sur la position initiale des agents sur le réseau. Cependant, la charge de chaque processus va dépendre de l'évolution de la population des agents évoluant sur la partie du réseau qu'il gère, ce qui va occasionner tôt ou tard un déséquilibre de charge global.

Les techniques de *scratch-remap* ne sont pas optimales pour équilibrer dynamiquement la charge de travail de notre simulation. Une simulation de trafic large échelle implique en effet la modélisation de graphe de taille importante. Le réseau de Paris-Saclay que nous utilisons dans nos expérimentations comporte par exemple 1856 sommets et 3784 arcs. Un redécoupage de l'environnement nécessite pour ce graphe une vingtaine de secondes à chaque fois, et occasionne de nombreux transferts d'agents. Comme il s'agit d'un système dynamique, nécessitant un rééquilibrage fréquent, ce type de solution est contre-productif.

C'est pourquoi nous avons développé un mécanisme d'équilibrage de charge dynamique, permettant de diffuser incrémentalement la charge excédante d'un processus aux processus avoisinants. Un processus A est dit connecté au processus B si il existe un arc du réseau routier connectant la partie de l'environnement gérée par A à la partie gérée par B. L'ensemble C_i des partitions connectées à P_i est définie comme suit :

$$C_i = \{P_j \mid \exists (v,u), v \in P_i, u \in P_j, i \neq j\} \tag{5.2}$$

Les sommets frontières d'une partition P_i avec la partition P_j, sont notées $F_{i,j}$:

$$F_{i,j} = \{v \in P_i \mid \exists (v,u), u \in P_j\} \tag{5.3}$$

$W(P_i)$ représente le nombre d'agents présents sur la partition P_i. Le poids d'un sommet est représenté par le nombre d'agents présents sur celui-ci (ainsi que sur les arcs partant de ce sommet).

Lorsque les processus ont fini l'exécution d'un pas de temps, et que les synchronisations inter-processus ont été effectuées, chacun d'eux vérifie le nombre d'agents présents sur la partie de l'environnement qu'il gère. Si ce nombre dépasse un certain seuil, le processus va sélectionner le processus connecté le moins chargé, et lui transférer le sommet de leur frontière (ainsi que les arcs sortants adjacents) ayant le plus d'agents (algorithme 7).

La clause de sélection du sommet $|v_{max}| < 0.5(n/k)$ est présente pour éviter qu'un sommet n'oscille constamment entre les processus si un trop grand nombre d'agents est présent sur ce sommet. La définition d'une limite supérieure d'agents pour le transfert empêche

Algorithm 7 Algorithme de diffusion de charge

Pré-conditions :P une partition d'un graphe $G = (V, E)$
Pré-conditions :P_i la partition courante
Pré-conditions :C_i l'ensemble des partitions connectées à P_i
Pré-conditions :n nombre total d'agents
Pré-conditions :k nombre de processus
$threshold \leftarrow \alpha(n/k)$
si $W(P_i) > threshold$ **alors**
  $P_{min} \leftarrow P_j \in C_i$ tel que $W(P_j)$ soit minimal
  $v_{max} \leftarrow$ le sommet le plus lourd $\in F_{i,min}$ avec $|v_{max}| < 0.5(n/k)$
  transférer v_{max} sur P_{min}
fin si

cet inconvénient. Un sommet plus léger sera choisi à sa place.

Le choix du coefficient α est crucial. Il détermine en effet la fréquence de déclenchement de la procédure d'équilibrage. Plus α sera proche de 1, plus souvent la procédure sera déclenchée.

Un exemple d'exécution de l'algorithme est illustré sur la figure 5.4. Dans cet exemple, la charge de chaque sommet est indiquée en son centre. La charge totale est de 100. Si l'on prend $\alpha = 1.2$, le processus d'équilibrage se déclenche lorsque la charge totale d'un processus dépasse 40. C'est le cas pour le processus 1 en (a). Ce processus détermine que le processus 2 s'avère être le moins chargé des processus connectés. Il sélectionne parmi les sommets qui sont frontaliers avec le processus 2 le sommet le plus lourd (b). Il transfère enfin ce sommet (et ses arcs sortant) à la partition numéro 2, pour obtenir une partition plus équilibrée.

5.5 Implémentation

La distribution des agents offre une distribution de la simulation optimale pour la simulation macroscopique. C'est pourquoi nous avons limité l'implémentation de notre algorithme de diffusion à la seule simulation microscopique.

Chaque processus tient à jour une liste des sommets qui sont à la frontière avec d'autres processus, ainsi qu'un annuaire indiquant la localisation des sommets. La procédure d'équilibrage est exécutée à chaque pas de temps, après que les différents processus se soient synchronisés. Les processus échangent alors des informations concernant leur charge de travail actuelle à l'aide d'un $MPI\_Allgather()$.

Si un processus dépasse le seuil de charge passé en paramètre, il lance la procédure de transfert. Il sélectionne le processus connecté le moins chargé, puis le sommet à la frontière de ce processus le plus lourd. Ce sommet est ensuite envoyé à l'aide d'un $MPI\_Send()$

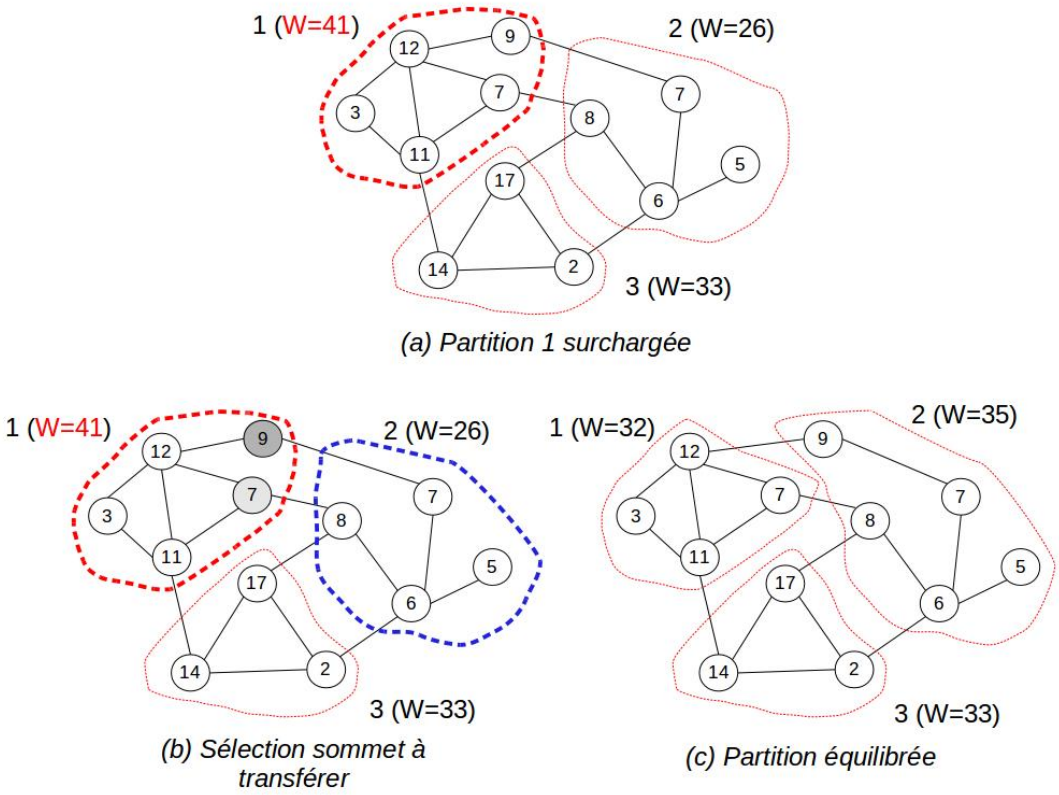

Figure 5.4: Déroulement de l'algorithme diffusif

qui sera reçu par un *thread* en écoute constante par le processus cible. Les sommets à la frontière sont ensuite mis à jour par les processus émetteur et récepteur.

Les agents présents sur le sommet transféré sont finalement déplacés vers le processus récepteur, à l'aide de la même procédure que celle utilisée pour la migration d'agents de processus en processus (séquence de $MPI\_Isend()$).

Une fois que tous les processus ont finalisé leur équilibrage, l'annuaire contenant l'emplacement des sommets est mis à jour avec un $MPI\_Allgather()$.

5.6 Résultats

5.6.1 Efficacité de l'équilibrage de charge

Nous avons testé notre algorithme de diffusion de charge sur une simulation de 1200 pas de temps du réseau Paris-Saclay. La simulation a été effectuée sur les 64 cœurs de calcul dont nous disposions. Les résultats présents dans cette partie sont issus d'une simulation

comportant 100000 agents.

Puisque l'efficacité de la simulation est limitée par le processus le plus lent, nous mesurons la qualité de la répartition de charge d'une partition comme étant la différence entre la charge du processus le plus chargé et la charge idéale. Dans notre cas, la charge d'un processus est égale au nombre d'agents présents sur celui-ci. La charge idéale est le nombre total d'agents divisé par le nombre de processus. Nous définissons la déviation maximale comme suit :

$$\Delta_P = W(P_{max}) - \frac{n}{k} \tag{5.4}$$

Ainsi, un Δ_P élevé signifie qu'un déséquilibre important est constaté. Nous avons testé différentes valeurs pour le paramètre α afin de déterminer laquelle est la plus efficace. La figure 5.5 compare l'évolution de Δ_P au cours du temps pour la version statique, et pour la version avec équilibrage de charge avec différentes valeurs de α.

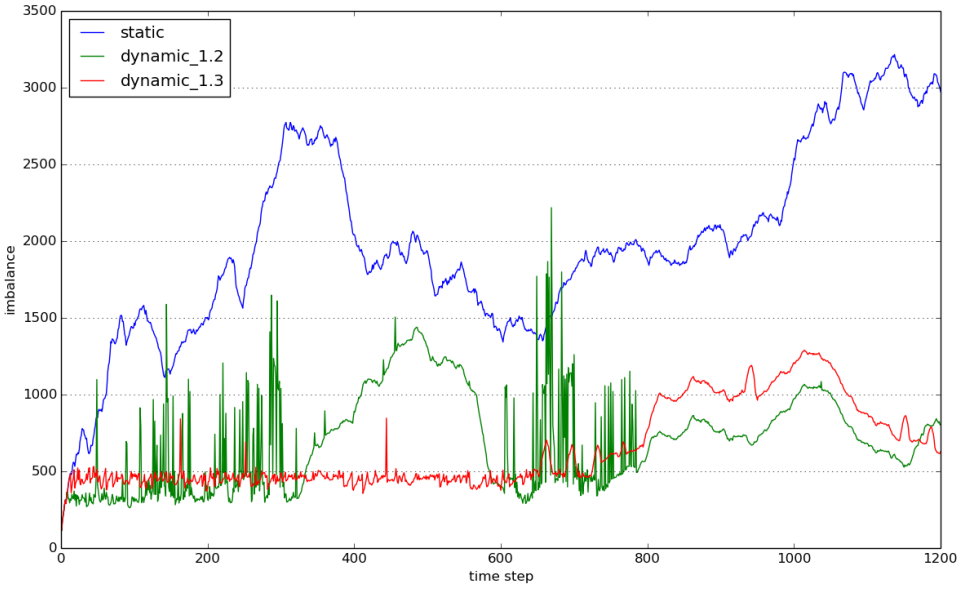

Figure 5.5: Évolution du déséquilibre de charge pour différentes configurations

Une valeur de α élevée impliquerait des partitions moins bien équilibrées, puisque le mécanisme d'équilibrage ne se déclencherait qu'avec un nombre d'agents très important. Cependant, comme nous pouvons le constater sur la figure 5.5, un α trop petit produit des partitions instables, avec une charge qui oscille constamment entre les unités de calcul. Ce phénomène est encore plus marqué avec $\alpha = 1.1$ (figure 5.6). À partir de $\alpha = 1.3$, les oscillations cessent, et la charge de la simulation est répartie efficacement entre les processus.

La hausse de Δ_P à la fin de la simulation est expliquée par le fait que le réseau est saturé, et que les partitions les plus chargées ne contiennent qu'un seul sommet très peuplé. La granularité de notre mécanisme d'équilibrage étant le sommet, il est impossible pour ce processus de diffuser sa charge plus avant.

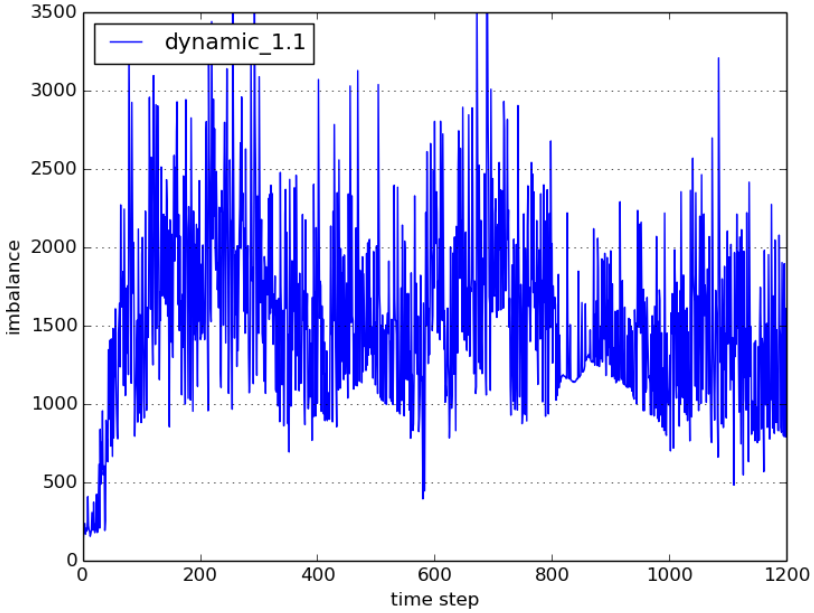

Figure 5.6: Oscillations causées par un α trop petit (ici $\alpha = 1.1$)

5.6.2 Performances

Nous avons comparé les performances de la simulation où l'environnement est distribué statiquement avec une simulation dont la charge est équilibrée dynamiquement. Dans notre simulateur de test, les agents ne font que se déplacer et calculer leurs chemins. Ce type d'agents est très rapide à exécuter, et un déséquilibre de charge ne provoque pas une différence de temps de calcul très important. Cependant, un simulateur réél peut être composé d'agents cognitifs beaucoup plus complexes que ceux que nous mettons en œuvre.

Pour simuler des comportements d'agents complexes, prenant du temps à s'exécuter, nous les avons fait attendre chacun 0.001 secondes à chaque fois qu'ils s'exécutent. Le tableau en figure 5.1 indique les temps d'exécution mesurés sur une simulation de 1000 pas de temps avec les deux méthodes (ainsi que la version séquentielle). La figure 5.7 montre les accélérations des méthodes de distribution relatives à l'exécution séquentielle de la même simulation.

Finalement, la figure 5.8 montre l'efficacité de l'équilibrage dynamique de l'environnement

nombre d'agents	10 000	50 000	100 000	250 000	500 000
Séquentiel (1 cœur)	12814	62672	142350	315876	631243
Statique (64 cœurs)	463	2136	3902	9636	18929
Dynamique_1.3 (64 cœurs)	327	1382	2665	6468	13480

Tableau 5.1: Temps d'exécution (en secondes) pour une simulation de 1000 pas de temps sur le réseau Paris-Saclay

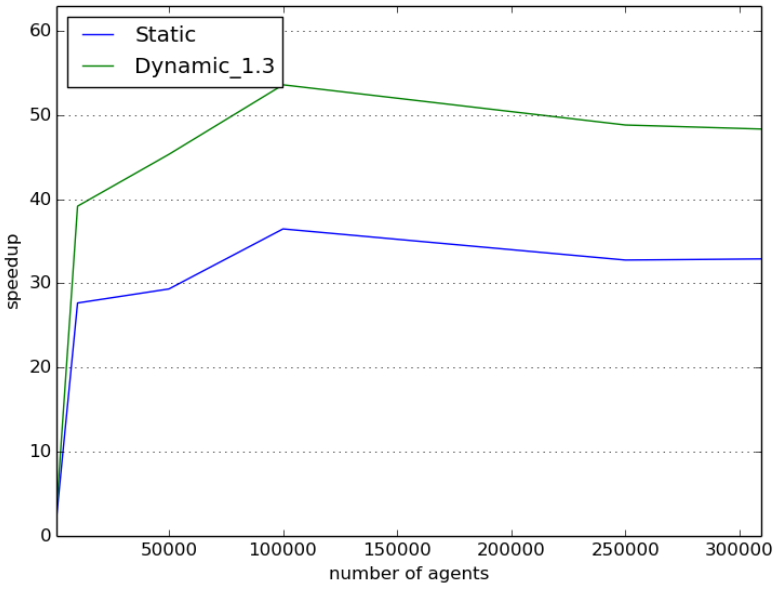

Figure 5.7: Accélération procurée par chaque méthode

en fonction du nombre de processus utilisés, pour une simulation de 1000 pas de temps faisant agir 100 000 agents. On peut voir que la simulation passe efficacement à l'échelle avec le nombre de processus que nous avons à disposition.

5.7 Conclusions et perspectives

Nous avons proposé dans ce chapitre un algorithme d'équilibrage de charge dynamique spécialement développé pour les simulations de trafic avec environnement distribué. Cet algorithme s'est révélé particulièrement efficace avec les conditions de test que nous avions : pour 100 000 agents, charge que l'on peut attendre sur le réseau Paris-Saclay, nous atteignons une accélération de 54 avec 64 processeurs. L'efficacité est de 0.8, ce qui se rapproche de l'optimal.

Ces résultats doivent cependant être nuancé. Notre environnement de test comporte deux

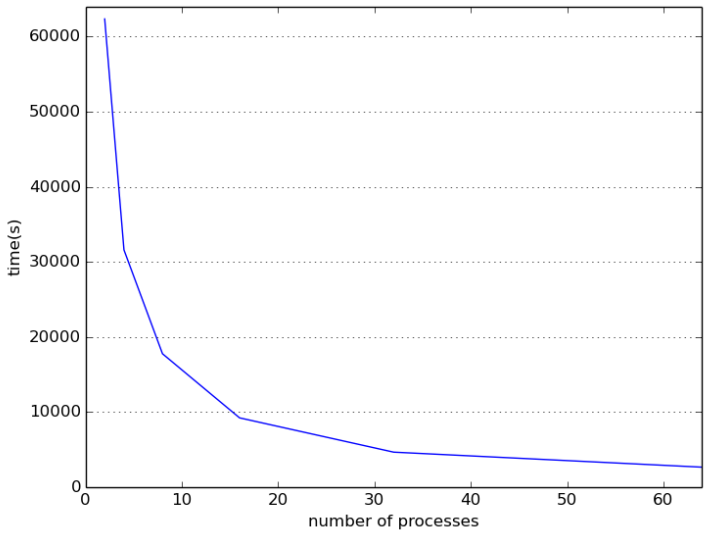

Figure 5.8: Temps d'exécution en fonction du nombre de processus

serveurs de 32 cœurs de calcul chacun. Les cœurs de calcul présents sur le même serveur disposent d'un vecteur de communication bien plus rapide que ce que l'on pourrait trouver sur un cluster de calcul traditionnel, où les processeurs sont le plus souvent reliés par un réseau ethernet. Il serait particulièrement intéressant de reproduire ces expérimentations sur un cluster réel.

En outre, bien que très efficace, notre algorithme d'équilibrage de charge diffusif est limité par sa granularité. Lorsqu'un processus doit gérer un sommet du graphe très chargé, il serait profitable de pouvoir distribuer les agents présents sur ce sommet entre plusieurs processus. Cette approche hybride sera étudiée dans de futurs travaux, et devrait encore améliorer l'efficacité de la distribution de l'environnement d'une simulation de trafic.

Nous avons par ailleurs envisagé une méthode permettant de limiter les migrations d'agents d'une UC à une autre par la mise en place d'un protocole de transfert prédictif de sommet. Dans certain cas, il est en effet possible d'éviter qu'un grand nombre d'agent ne soit transférer d'une partition à une autre en rapatriant préventivement leur sommet de destination sur leur partition courante.

Par exemple dans la figure 5.9, l'épaisseur des arcs est d'autant plus grande que le nombre d'agents présent sur ces arcs est important. Les différentes couleurs symbolisent deux partitions. On voit en (a) qu'un grand nombre d'agents se déplacent de la partition grise vers la partition marron. Ils vont devoir être transférés dès lors qu'ils atteindront le sommet 3. Si l'on déplace le sommet 3 vers la partition grise (b), toutes ces migrations seront évitées. L'étude de cette approche sera l'objet de travaux futurs.

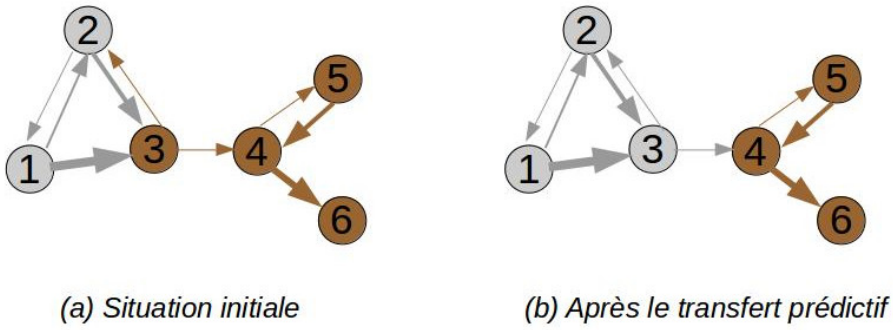

Figure 5.9: Transfert prédictif de sommets

Conclusions et travaux futurs

Bilan

Ce travail de thèse a été particulièrement motivé par la proposition de solutions permettant la simulation du trafic routier à grande échelle.

Ce mémoire a présenté plusieurs outils permettant la distribution de différents modèles de simulateurs de trafic multi-agent sur un système distribué. Les résultats obtenus par l'expérimentation ont montré l'efficacité de ces techniques, qui peuvent, avec quelques ajustements mineurs, être directement mises en œuvre sur les simulateurs existants. Ceux-ci pourront grâce à nos travaux être exécutés sur des infrastructures de haute performance.

Les contributions apportées au cours de ce travail de thèse se situent aussi bien sur un point de vue théorique qu'applicatif.

- Nous avons développé un modèle générique de simulateur de trafic, réunissant les caractéristiques fondamentales des simulateurs de la littérature, qu'ils soient microscopiques ou macroscopiques.
- Nous avons proposé deux méthodes permettant la distribution de ce simulateur sur de multiples cœurs de calcul. La distribution des agents dispose d'une charge parfaitement équilibrée, tandis que la distribution de l'environnement est destinée à limiter les communications entre les unités de calcul.
- Nous fournissons en détail un protocole permettant d'implémenter ces méthodes. Les technologies utilisées sont des standards industriels, ce qui facilite la réutilisation de nos travaux dans un contexte opérationnel.
- Nous avons testé chacune des méthodes présentées sur des simulations macroscopiques et microscopiques. Nous avons critiqué ces résultats, afin de déterminer quelle méthode est la mieux adaptée pour chaque situation.
- Nous avons développé un algorithme permettant d'équilibrer dynamiquement la charge de chaque unité de calcul, améliorant considérablement les performances de la simulation microscopique.

Perspectives

Les futurs travaux de recherche ouverts par ce travail de thèse porteront sur deux aspects. Le premier sera l'élargissement des conditions expérimentales, permettant de confirmer les résultats obtenus ici :

- Par la mise en place d'un cluster réunissant un grand nombre de machines, permettant d'une part de tester le passage à l'échelle des méthodes avec le nombre de machines, et d'autre part de mesurer plus précisément l'impact des communications inter-machines sur un réseau ethernet.
- Par l'implémentation des algorithmes proposés sur les simulateurs les plus utilisés, afin de valider leur efficacité dans un contexte opérationnel.

Le deuxième aspect se concentrera plus particulièrement sur l'extension des solutions proposées afin d'en améliorer encore les performances.

- Par l'affinage du grain de l'algorithme d'équilibrage de charge diffusif, grâce à la mise en place d'une méthode hybride, permettant de distribuer des agents présents sur un même élément du graphe entre plusieurs unités de calcul.
- Par l'implémentation et le test d'un transfert prédictif de sommets, comme évoqué dans la partie 5.7.

Ce travail de thèse a fait l'objet des publications suivantes:

M. Mastio, M. Zargayouna, G. Scemama & O. Rana, "Distributed agent-based traffic simulations", IEEE Intelligent Transportation Systems Magazine, IEEE Computer Society, 10 pages, 2017 [61]

M. Mastio, M. Zargayouna, G. Scemama & O. Rana, "Modèles de distribution des simulations multi-agents de mobilité des voyageurs", Technique et science informatiques – n°6/2016, Lavoisier, pp. 675-694, 2016 [60]

M. Mastio, M. Zargayouna, O.. Rana, G. Scemama, "Patterns to distribute mobility simulations", IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA), 10 pages, IEEE Computer Society, 2016 [59]

M. Mastio, M. Zargayouna, O. Rana, "Towards a Distributed Multiagent Travel Simulation", In Smart Innovation, Systems and Technologies, vol. 38, pp. 15-25, Springer International Publishing, KES-AMSTA 2015 [57]

M. Mastio, M. Zargayouna, O. Rana, G. Scemama, "Méthodes de distribution pour les simulations de mobilité des voyageurs", 23èmes Journées Francophones sur les Systèmes Multi-Agents (JFSMA'15), pp. 175-184, Cépaduès, 2015. [58]

Bibliographie

[1] Afshin Abadi, Tooraj Rajabioun, and Petros A Ioannou. Traffic flow prediction for road transportation networks with limited traffic data. *IEEE Transactions on Intelligent Transportation Systems*, 16(2):653–662, 2015.

[2] Md Salman Ahmed and Mohammad A Hoque. Partitioning of urban transportation networks utilizing real-world traffic parameters for distributed simulation in sumo. In *Vehicular Networking Conference (VNC), 2016 IEEE*, pages 1–4. IEEE, 2016.

[3] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[4] E.S. Angelotti, E.E. Scalabrin, and B.C. Avila. PANDORA: a multi-agent system using paraconsistent logic. In *Fourth International Conference on Computational Intelligence and Multimedia Applications, 2001. ICCIMA 2001. Proceedings*, pages 352–356, 2001. doi: 10.1109/ICCIMA.2001.970493.

[5] Fabien Badeig, Flavien Balbo, Gérard Scemama, and Mahdi Zargayouna. Agent-based coordination model for designing transportation applications. In *Intelligent Transportation Systems, 2008. ITSC 2008. 11th International IEEE Conference on*, pages 402–407. IEEE, 2008.

[6] Jaime Barceló and Jordi Casas. Dynamic network simulation with aimsun. In *Simulation approaches in transportation analysis*, pages 57–98. Springer, 2005.

[7] Stephen T. Barnard and Horst D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Pract. Exper.*, 6(2):101–117, April 1994. ISSN 1096-9128. doi: 10.1002/cpe.4330060203.

[8] Roberto Battiti and Alan A Bertossi. Differential greedy for the 0-1 equicut problem. In *Network Design: Connectivity and Facilities Location*, pages 3–22, 1997.

[9] Roberto Battiti and Alan A. Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, 1999.

[10] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. SUMO - simulation of urban MObility - an overview. In *SIMUL 2011, The Third International Conference on Advances in System Simulation*, pages 55–60, 2011. ISBN 978-1-61208-169-4.

[11] F Bellifemine, A Poggi, and G Rimassa. {JADE}: A {FIPA}-compliant agent framework. pages 97–108.

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

[13] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[14] Rupak Biswas and David Bailey. Parallel load balancing for adaptive unstructured meshes. 1998.

[15] Dietrich Braess, Anna Nagurney, and Tina Wakolbinger. On a paradox of traffic planning. *Transportation science*, 39(4):446–450, 2005.

[16] Michael E Bratman, David J Israel, and Martha E Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 1988.

[17] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42(3):153–159, May 1992. ISSN 0020-0190. doi: 10.1016/0020-0190(92)90140-Q.

[18] Alexis Champion, Stéphane Éspié, and Jean-Michel Auberlet. Behavioral road traffic simulation with archisim. In *Summer Computer Simulation Conference*, pages 359–364. Society for Computer Simulation International; 1998, 2001.

[19] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough. 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 (HPCC-ICESS)*, pages 538–545, 2012. doi: 10.1109/HPCC.2012.79.

[20] Nicholson Collier and Michael North. Repast HPC: A platform for large-scale agent-based modeling. In Werner Dubitzky, Krzysztof Kurowski, and Bernhard Schott, editors, *Large-Scale Computing*, pages 81–109. John Wiley & Sons, Inc., 2011. ISBN 9781118130506.

[21] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. A framework for distributing agent-based simulations. In *European Conference on Parallel Processing*, pages 460–470. Springer, 2011.

[22] Gennaro Cordasco, Francesco Milone, Carmine Spagnuolo, and Luca Vicidomini. Exploiting d-mason on parallel platforms: a novel communication strategy. In *European Conference on Parallel Processing*, pages 407–417. Springer, 2014.

[23] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE concurrency*, 7(1):22–31, 1999.

[24] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[25] Carlos F Daganzo. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Transportation Research Part B: Methodological*, 28(4):269–287, 1994.

[26] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing, January 19 2010. US Patent 7,650,331.

[27] Yunhua Deng and Rynson WH Lau. On delay adjustment for dynamic load balancing in distributed virtual environments. *IEEE transactions on visualization and computer graphics*, 18(4):529–537, 2012.

[28] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[29] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM J. Res. Dev.*, 17(5):420–425, September 1973. ISSN 0018-8646. doi: 10.1147/rd.175.0420.

[30] Derek L Eager, Edward D Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1): 53–68, 1986.

[31] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[32] J Ferber. Les systemes multi-agents, vers une intelligence collective, intereditions, or multi-agent systems, an introduction to distributed artificial intelligence, 1995.

[33] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Conference on Design Automation, 1982*, pages 175–181, June 1982. doi: 10.1109/DAC.1982.1585498.

[34] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[35] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Wiley New York, 2000.

[36] Peter G Gipps. A behavioural car-following model for computer simulation. *Transportation Research Part B: Methodological*, 15(2):105–111, 1981.

[37] Maxime Gueriau, Romain Billot, Nour-Eddin El Faouzi, Salima Hassas, and Frédéric Armetta. Multi-Agent Dynamic Coupling for Cooperative Vehicles Modeling. In 30/01/2015, editor, *The Twenty-Ninth Conference on Artificial Intelligence AAAI'2015 - (DEMO Track)*, January 2015.

[38] Frank A Haight. *Mathematical theories of traffic flow*. 1965.

[39] Dirk Helbing, Ansgar Hennecke, Vladimir Shvetsov, and Martin Treiber. Master: macroscopic traffic simulation based on a gas-kinetic, non-local traffic model. *Transportation Research Part B: Methodological*, 35(2):183–211, 2001.

[40] Ta-Yin Hu, Chee-Chung Tong, Tsai-Yun Liao, and Wei-Ming Ho. Simulation-assignment-based travel time prediction model for traffic corridors. *IEEE Transactions on Intelligent Transportation Systems*, 13(3):1277–1286, 2012.

[41] Jan Hueper, Gunes Dervisoglu, Ajith Muralidharan, Gabriel Gomes, Roberto Horowitz, and Pravin Varaiya. Macroscopic modeling and simulation of freeway traffic flow. *IFAC Proceedings Volumes*, 42(15):112–116, 2009.

[42] Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38, 1998.

[43] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, January 1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997.

[44] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1995.

[45] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[46] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1970.tb01770.x.

[47] Daniel Krajzewicz, Georg Hertkorn, Christian Rössel, and Peter Wagner. Sumo (simulation of urban mobility)-an open-source traffic simulation. In *Proceedings of the 4th middle East Symposium on Simulation and Modelling (MESM20002)*, pages 183–187, 2002.

[48] Alban Rousset Benedicte Herrmann Christophe Lang and Laurent Philippe. A survey on parallel and distributed multi-agent systems for high performance computing.

[49] Hans Petter Langtangen and Xing Cai. On the efficiency of python for high-performance computing. In *Modeling, Simulation and Optimization of Complex Processes*, pages 337–357. Springer Berlin Heidelberg, January 2008. ISBN 978-3-540-79408-0, 978-3-540-79409-7.

[50] Guillaume Laville, Kamel Mazouzi, Christophe Lang, Nicolas Marilleau, and Laurent Philippe. Using gpu for multi-agent multi-scale simulations. In *Distributed computing and artificial intelligence*, pages 197–204. Springer, 2012.

[51] Kyungmin Lee and Dongman Lee. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 160–168. ACM, 2003.

[52] Jie Li and Hisao Kameda. Load balancing problems for multiclass jobs in distributed/-parallel computer systems. *IEEE Transactions on Computers*, 47(3):322–332, 1998.

[53] Michael J Lighthill and Gerald Beresford Whitham. On kinematic waves. ii. a theory of traffic flow on long crowded roads. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 229, pages 317–345. The Royal Society, 1955.

[54] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, July 2005.

[55] Charles M Macal and Michael J North. Agent-based modeling and simulation. In *Winter simulation conference*, pages 86–98. Winter simulation conference, 2009.

[56] René Mandiau, Alexis Champion, Jean-Michel Auberlet, Stéphane Espié, and Christophe Kolski. Behaviour based on decision matrices for a coordination between agents in a urban traffic simulation. *Applied Intelligence*, 28(2):121–138, 2008.

[57] Matthieu Mastio, Mahdi Zargayouna, Gérard Scemama, and Omer Rana. Towards a distributed multiagent travel simulation. In *In Smart Innovation, Systems and Technologies, Springer International Publishing, KES-AMSTA 2015*, volume 38, pages 15–25, 2015.

[58] Matthieu Mastio, Mahdi Zargayouna, Gérard Scemama, and Omer Rana. Méthodes de distribution pour les simulations de mobilité des voyageurs. In *23èmes Journées Francophones sur les Systèmes Multi-Agents (JFSMA 15)*, pages 175–184, 2015.

[59] Matthieu Mastio, Mahdi Zargayouna, Gérard Scemama, and Omer Rana. Patterns to distribute mobility simulations. In *IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, 2016.

[60] Matthieu Mastio, Mahdi Zargayouna, Gérard Scemama, and Omer Rana. Modèles de distribution des simulations multi-agents de mobilité des voyageurs. *Technique et science informatiques, Lavoisier*, (6):675–694, 2016.

[61] Matthieu Mastio, Mahdi Zargayouna, Gérard Scemama, and Omer Rana. Distributed agent-based traffic simulations. *IEEE Intelligent Transportation Systems Magazine, IEEE Computer Society*, 2017.

[62] Albert Messner and Markos Papageorgiou. Metanet: A macroscopic simulation program for motorway networks. *Traffic Engineering & Control*, 31(8-9):466–470, 1990.

[63] Maciejewski Michal and Nagel Kai. Towards multi-agent simulation of the dynamic vehicle routing problem in matsim. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II*, PPAM'11, pages 551–560, Berlin, Heidelberg, 2012. Springer-Verlag.

[64] Fabien Michel. Intégration du calcul sur gpu dans la plate-forme de simulation multi-agent générique turtlekit 3. In *JFSMA: Journées Francophones sur les Systèmes Multi-Agents*, pages 135–144, 2013.

[65] Naoki Miyata and Toru Ishida. Community-based load balancing for massively multi-agent systems. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 28–42. Springer, 2007.

[66] Pedro Morillo, Juan M Orduna, Marcos Fernandez, and Jose Duato. Improving the performance of distributed virtual environment systems. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):637–649, 2005.

[67] Kai Nagel and Marcus Rickert. Parallel implementation of the transims micro-simulation. *Parallel Computing*, 27(12):1611–1639, 2001.

[68] Kai Nagel, Paula Stretz, Martin Pieck, Rick Donnelly, and Christopher L Barrett. Transims traffic flow characteristics. *arXiv preprint adap-org/9710003*, 1997.

[69] Beatrice Ng, Antonio Si, Rynson WH Lau, and Frederick WB Li. A multi-server architecture for distributed virtual walkthrough. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 163–170. ACM, 2002.

[70] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):3, 2013-03-13. ISSN 2194-3206. doi: 10.1186/2194-3206-1-3.

[71] Leonid Oliker and Rupak Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[72] Omar Rihawi. *Modelling and simulation of distributed large scale situated multi-agent systems*. PhD thesis, Université Lille1, 2014.

[73] Omar Rihawi, Yann Secq, and Philippe Mathieu. Relaxing synchronization constraints in distributed agent-based simulations. *Jurnal Teknologi (Sciences and Engineering)*, 63(3):65–76, 2013.

[74] Omar Rihawi, Yann Secq, and Philippe Mathieu. Effective distribution of large scale situated agent-based simulations. In *ICAART 2014 6th International Conference on Agents and Artificial Intelligence*, volume 1, pages 312–319. SCITEPRESS Digital Library, 2014. ISBN 9978-989-758-015-4.

[75] Omar Rihawi, Yann Secq, and Philippe Mathieu. Load-balancing for large scale situated agent-based simulations. *Procedia Computer Science*, 51:90–99, 2015.

[76] Thomas C Schelling. Dynamic models of segregation. *Journal of mathematical sociology*, 1(2):143–186, 1971.

[77] Kirk Schloegel, George Karypis, and Vipin Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 59–59. IEEE, 2000.

[78] Kirk Schloegel, George Karypis, and Vipin Kumar. Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.

[79] David Strippgen and Kai Nagel. Multi-agent traffic simulation with cuda. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 106–114. IEEE, 2009.

[80] Katia P Sycara. Multiagent systems. *AI magazine*, 19(2):79, 1998.

[81] Anthony Ventresque, Quentin Bragard, Elvis S Liu, Dawid Nowak, Liam Murphy, Georgios Theodoropoulos, and Qi Liu. Spartsim: A space partitioning guided by road network for distributed traffic simulations. In *Proceedings of the 2012 IEEE/ACM 16th international symposium on distributed simulation and real time applications*, pages 202–209. IEEE Computer Society, 2012.

[82] Guillermo Vigueras, Miguel Lozano, Juan Manuel Orduña, and Francisco Grimaldo. A comparative study of partitioning methods for crowd simulations. *Applied Soft Computing*, 10(1):225–235, 2010.

[83] Chris Walshaw. Multilevel refinement for combinatorial optimisation: Boosting metaheuristic performance. In *Hybrid Metaheuristics*, number 114 in Studies in Computational Intelligence, pages 261–289. Springer Berlin Heidelberg, January 2008. ISBN 978-3-540-78294-0, 978-3-540-78295-7.

[84] Uri Wilensky. Netlogo. 1999.

[85] Marc H Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on parallel and distributed systems*, 4(9):979–993, 1993.

[86] Mahdi Zargayouna, Besma Zeddini, Gérard Scemama, and Amine Othman. Simulating the impact of future internet on multimodal mobility. In *The 11th ACS/IEEE International Conference on Computer Systems and Applications AICCSA'2014*. IEEE Computer Society, 2014.