RenPar'2000
Besançon, 19 juin - 22 juin 2000

Bibliothèque parallèle pour l'implantation de systèmes multi-agent à composantes connexionnistes

Eugen Dedu

Supélec, campus de Metz
2 rue Edouard Belin
57070 Metz

Résumé
Les réseaux de neurones, par leur généralité, peuvent apporter des solutions à des problèmes difficiles à caractériser. La modélisation des systèmes multi-agent utilise des entités qui doivent s'adapter eux-mêmes dans leur environnement, parfois inconnu. Pour essayer différents comportements des agents (réactif, cognitif), les chercheurs ont besoin de créer rapidement leurs applications. Ces applications, qui nécessitent parfois des tailles énormes de données, doivent être rapides à l'exécution, sans pour autant complexifier leur implantation. Cet article propose une bibliothèque pour des machines parallèles à mémoire partagée, qui permet d'implanter simplement une catégorie de systèmes multi-agent où les agents peuvent utiliser des réseaux de neurones dans leur comportement cognitif. L'accent est mis sur les problèmes posés à la parallélisation.
Mots-clés : Parallélisme, Algorithmique, Performances, Systèmes multi-agent.

---

1. Motivations

Les réseaux de neurones et les systèmes multi-agent sont deux sous-domaines de l'intelligence artificielle. Le premier fait un rapprochement avec le cerveau humain. Le deuxième utilise le concept d'agent, qui est défini par Jennings & Wooldridge [6, page 4] comme une entité située dans un certain environnement, capable d'actions autonomes pour atteindre ses objectifs. Un agent *situé* est un agent qui modélise une entité physique, soumise aux contraintes spatiales.

Le paradigme multi-agent est mieux adapté pour certains problèmes, comme la simulation d'une société où les agents peuvent être des hommes, des véhicules etc. L'accent est ainsi mis sur la coopération de ces agents. Certains auteurs considèrent ce paradigme comme un pas en avant dans l'histoire de la programmation [5, 7].

La bibliothèque présentée dans cet article a pour but de faciliter l'écriture des systèmes de type ParMASS [8, 2] (Par*allel* M*ulti-*A*gent* S*ystem* S*imulator*) et d'optimiser leur exécution (figure 1). Ces applications mettent en jeu une large population d'agents sur un monde à grande taille. Leurs buts sont :

- vérifier l'extensibilité des algorithmes utilisés dans les systèmes multi-agent ;
- mettre au point des comportements efficaces pour les agents ;
- chercher des phénomènes émergents — des comportements sociaux à partir de comportements individuels.

Un exemple typique est un système multi-agent situé dans un environnement discret avec des obstacles, où les agents ont comme but de ramener tout le minerai des mines dans les usines. À ce jour, leur comportement est essentiellement réactif. ParMASS-2 devrait permettre l'introduction des réseaux de neurones dans le comportement des agents. ParMASS-2 étant encore à l'étude, la bibliothèque essaie d'être riche en expression (figure 2). Une autre application est donnée dans [3] : dans un environnement discret bidimensionnel, le but des agents est de mettre des objets dans une certaine configuration.

Le parallélisme intrinsèque de ces applications et les grands temps de simulation en font de bons candidats aux machines parallèles. Cependant, plusieurs problèmes rendent difficile leur parallélisation, comme l'équilibrage de charge entre les processeurs, la bonne utilisation des caches (*contention* [9] et

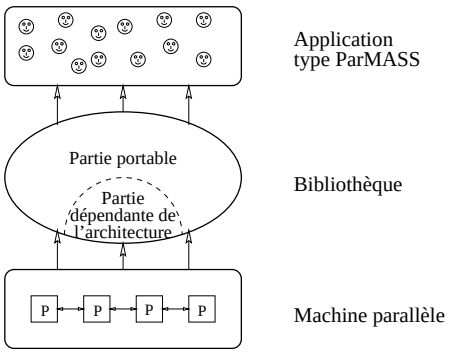

FIG. 1 – *La bibliothèque parallélise des applications de type ParMASS*

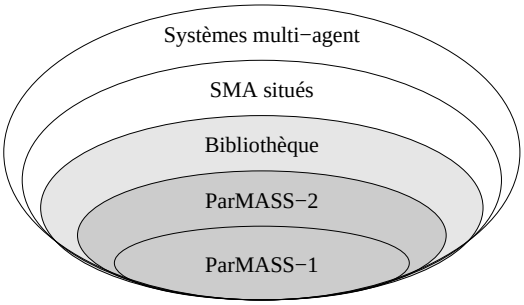

FIG. 2 – *ParMASS est un système multi-agent situé*

false-sharing [1] de la mémoire), l'aspect fortement dynamique et le grain fin de l'application.

Typiquement, lors de l'etude d'un tel problème, plusieurs solutions sont essayées, et chacune est implantée et exécutée seulement quelques fois. Ce point impose la *simplicité d'implantation* sur le premier plan, tout en essayant d'avoir de bonnes performances et de la richesse en expression.

2. La bibliothèque

Pour simuler *facilement* des systèmes de type ParMASS une bibliothèque a été créée. D'autres choix sont soit beaucoup plus difficiles à concevoir, implanter et maintenir (compilateur avec un langage de programmation), soit plus limités ou difficiles (translateur source-source). La bibliothèque offre des algorithmes (propagation du potentiel des ressources, calcul des cases visibles en présence d'obstacles), la parallélisation quasi-transparente à l'utilisateur final, la minimisation du *false-sharing*, des séquences de nombres aléatoires (et donc des résultats) identiques, indépendamment du nombre de threads. Elle permet également [2] de calculer le temps processeur passé dans un agent, dans les barrières ou dans d'autres parties du programme (en utilisant des timers précis : les compteurs internes aux processeurs).

2.1. Le type de parallélisme utilisé par la bibliothèque

À cause du grain fin des applications, les machines à mémoire distribuée ne sont pas appropriées. Les machines envisagées pour cette bibliothèque sont les machines à mémoire partagée (multi-processeurs). Ceci comprend les machines SMP [3] [9], où l'accès à la mémoire est uniforme, et CC-NUMA [4] [4], où le temps d'accès à la mémoire dépend de l'adresse.

L'application ParMASS est constituée de parties qui communiquent fortement. La parallélisation par processus ou par envoi de messages, qui ne partagent par défaut leur espace d'adressage, semble ainsi ne pas être appropriée. Deux autres méthodes de parallélisation sont spécifiques aux multi-processeurs : par threads ou par directives de parallélisation.

- Les threads représentent le niveau le plus bas de parallélisation. Plusieurs bibliothèques *multi-threading* existent à ce jour, et un effort de standardisation a été fait : la bibliothèque *pthreads* [10]. Leur richesse d'expression devient utile pour des applications à grain fin.
- Les directives de parallélisation sont beaucoup plus faciles à utiliser et offrent diverses fonctionnalités complexes d'une façon simple, comme le calcul automatique du nombre de threads ou la création automatique des variables privées aux threads. Cette méthode permet une parallélisation incrémentale, en partant du code séquentiel, facilitant ainsi la programmation et la mise au point des programmes. OpenMP en est un exemple, qui est aussi portable. Cependant, certaines facilités importantes d'OpenMP ne sont pas utiles dans cette bibliothèque : la bibliothèque étant optimisée pour des applications à grain fin, un très bon contrôle sur la parallélisation est nécessaire. Par

1. Le partage d'une même ligne de cache par deux variables, qui peut donner lieu à des conflits de cache
2. Partie encore à l'étude
3. *Symmetric MultiProcessor*
4. *Cache-Coherence, Non-Uniform Memory Access*

exemple, les conflits de cache étant très pénalisants, le même nombre de threads doit être utilisé pendant toute la simulation et le partitionnement doit être identique.

La bibliothèque est écrite en *C*. De nos jours, les architectures parallèles continuent à évoluer rapidement. La portabilité d'un logiciel devient importante. Pour l'instant, la parallélisation a été faite en *pthreads* et en Irix-native threads. Environ 10 % du code source sont dépendants de la bibliothèque *multi-threading* (figure 1), la parallélisation avec d'autres bibliothèques n'étant pas difficile. La parallélisation dans la version actuelle est complètement transparente pour l'utilisateur, sauf la spécification du nombre de threads. En séquentiel, elle peut être utilisée sur toute plate-forme avec un compilateur *C* ISO (ANSI).

2.2. Spécifications

Les parties principales d'une application écrite à l'aide de cette bibliothèque sont :

- L'*arbitre* : c'est la partie qui vérifie que les lois de l'univers et certaines lois globales sont satisfaites. Il est la seule partie qui a des connaissances globales et qui peut par exemple décider de finir l'application ou résoudre les conflits entre les agents (voir plus bas).
- L'*environnement* : il est bidimensionnel et discrétisé en espace. Il est composé de cases libres, d'obstacles et de ressources. Plusieurs types de ressources peuvent exister (sources et capteurs de minerai par exemple). Pour connaître la position des ressources, les agents utilisent un ou plusieurs de leurs percepts. Le potentiel qu'elles propagent dans l'environnement est une fonction de leur charge. Cette propagation est déjà implantée par un algorithme de propagation par vagues (voir section 2.5).
- Les *agents* : un agent occupe une case de l'environnement. À chaque cycle, un agent effectue une action parmi plusieurs, comme rester sur place, se déplacer dans une case voisine, prendre ou déposer du minerai. Les agents peuvent avoir des vitesses différentes. Leur comportement (réactif et/ou cognitif) est décrit par une fonction, qui peut utiliser des réseaux de neurones. Comme percepts des agents, l'application ParMASS utilise la perception du potentiel, la vision et des marques sur l'environnement. La communication entre les agents et la génétique des agents sont encore à l'étude.
- L'*initialisation* du système peut se faire aléatoirement ou de façon précise, en code source ou à partir de fichiers de configuration. Pour pouvoir être changés séparément et pour une initialisation plus rapide (en parallèle), les fichiers de configuration peuvent être liés seulement à une zone de l'environnement ; dans ce cas le système est initialisé à partir de plusieurs fichiers.
- Les *sauvegardes* sont de deux types : partielles ou totales. Les sauvegardes partielles utilisent une partie des données du système (l'évolution d'un certain agent par exemple). Les sauvegardes totales utilisent *tout* l'état du système à un instant donné.
- La *fin* de la simulation peut apparaître quand il n'y a plus d'agents ou quand l'arbitre décide cela, par exemple quand un certain nombre de cycles sont passés.

La méthodologie de conception d'une application utilisant la bibliothèque est :

- écrire soit une partie d'initialisation du système (en faisant appel aux fonctions fournies dans la bibliothèque), soit des fichiers de configuration ;
- écrire une fonction (la *fonction-utilisateur*) qui est appelée à la fin de chaque cycle (voir plus bas), permettant de finir la simulation, faire des sauvegardes, afficher diverses informations etc.
- pour chaque type de ressource, écrire la fonction permettant de calculer son potentiel en fonction de sa charge ;
- pour chaque type d'agent, écrire la fonction de comportement.

La simulation est discrétisée en temps ; à chaque cycle, la bibliothèque réalise, dans l'ordre, les opérations suivantes :

- appelle la fonction de comportement de chaque agent ;
- résout tous les conflits spatiaux (plusieurs agents essaient d'entrer dans la même case) : seul l'agent gagnant fait son mouvement, les autres restent sur place ;
- exécute les mouvements des agents ;
- appelle la fonction-utilisateur ;
- rafraîchit, si besoin, le champs de potentiels des ressources.

2.3. Le partitionnement

Les parties à paralléliser dans une application de type ParMASS sont les agents et l'environnement (comme la propagation des potentiels). À cause des conflits de caches qui peuvent apparaître, et qui

dégradent les performances d'une application à grain fin, le même partitionnement (pour les agents et pour les ressources) a été choisi. Deux solutions apparaissent ainsi :

- partitionnement de données, où le partitionnement est fait en fonction du nombre des données ;
- partitionnement de domaine, où chaque thread gère une maille de l'environnement.

La première solution n'est pas convenable : les agents changeant de place dans l'environnement, des conflits de cache apparaissent pour les agents se trouvant proches l'un de l'autre.

La deuxième solution a donc été utilisée. L'environnement est partitionné en plusieurs mailles de forme rectangulaire (figure 3). Actuellement, les mailles ont la largeur égale à la largeur de l'environnement, mais d'autres choix (pour minimiser la taille des frontières ou les échanges aux frontières) sont à l'étude.

Pour minimiser les conflits de cache, le même partitionnement a donc été choisi pour l'environnement et pour les agents. Pour les agents, cela signifie que des agents peuvent changer de zone, donc être traités par des threads différents dans des moments différents de temps. En contrepartie, chaque conflit spatial est traité par un seul thread. Pour la propagation des potentiels, cela signifie la gestion des champs de potentiel qui couvrent plusieurs mailles. Des barrières de synchronisation sont nécessaires, jusqu'à la propagation entière des potentiels.

Pour équilibrer la charge des processeurs, un partitionnement dynamique est à l'étude, qui permet de modifier la taille des mailles en fonction du temps processeur utilisé pour leur gestion.

2.4. Nombres aléatoires en parallèle

Les nombres aléatoires sont fréquemment utilisés dans les applications. Il est souvent nécessaire d'avoir les mêmes résultats pour différentes exécutions d'une application, même si le nombre de threads utilisés est différent d'une exécution à l'autre. Les bibliothèques système actuelles offrent des générateurs de nombres aléatoires donnant la même séquence, et il est possible d'avoir plusieurs tels générateurs dans une même application.

Une solution qui peut s'adapter à ParMASS est d'avoir un *seed*[5] associé à chaque case de l'environnement. Cela ayant besoin de beaucoup de mémoire, une autre solution a été envisagée. En fonction du nombre d'agents concernés, les nombres aléatoires apparaissent en trois situations :

- aucun agent n'est concerné, qui apparaît à l'initialisation du système, pour poser les parties de l'environnement, comme les ressources ou les obstacles ;
- un seul agent est concerné, qui apparaît en cas de mouvement aléatoire d'un agent (utile s'il ne perçoit aucune information sur les ressources) ;
- plusieurs agents sont concernés : ce cas apparaît en conflits.

La solution actuelle utilise un *seed* par agent. L'unicité des séquences est assurée par un *seed* donné par l'utilisateur de la bibliothèque, qui est utilisé pour initialiser le *seed* des agents et pour l'initialisation du système. En cas de conflits, le *seed* de l'agent d'indice maximum est utilisé.

2.5. L'algorithme de propagation par vagues

Cet algorithme est utilisé en ParMASS pour propager le potentiel des ressources. Les agents sont ainsi guidés vers les ressources. Quelques tests ont cependant révélé qu'il peut utiliser une grande partie du temps d'exécution séquentielle d'une application. Une implantation efficace est donc nécessaire. Cette section présente succinctement trois méthodes d'implantation de cet algorithme en séquentiel.

Cet algorithme fait intervenir différents types de ressources de potentiel et des agents qui ont besoin de connaître la valeur de ces potentiels dans certaines cases. Pour simplifier les calculs, un seul type de ressource est désormais supposé. Le potentiel dans une case est la valeur maximum entre :

1. $potentiel(x) - 1$, où x est chacune des quatre cases voisines (on suppose que le potentiel se propage seulement en ligne droite, pas en diagonale) ;
2. si la case contient une ressource, le potentiel donnée par cette ressource.

Dans un environnement sans obstacles, le nombre de cases où le potentiel d'une ressource est non vide est :

$$1 + 4 * \sum_{i=1}^{n-1} i = 2n^2 - 2n + 1 \quad (1)$$

où n représente le potentiel dans la case ressource (figure 4).

5. Le premier nombre d'une séquence de nombres aléatoires. Si le *seed* est identique, la même séquence est générée.

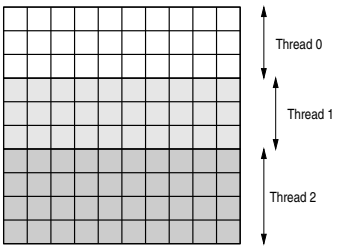

FIG. 3 – *L'environnement est partitionné en rectangles, chacun géré par un thread*

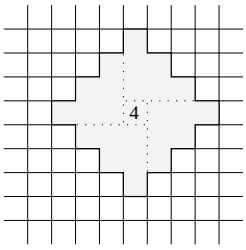

FIG. 4 – *Sans obstacles, le potentiel d'une ressource s'étend sur* $2n^2 - 2n + 1$ *cases (ici,* $n = 4$*)*

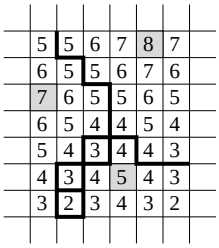

FIG. 5 – *Chaque ressource impose son potentiel sur un groupe de cases*

Pour implanter cet algorithme, plusieurs solutions existent :

1. La solution la plus simple est de rafraîchir tout l'environnement à chaque cycle où le potentiel des ressources change. La bibliothèque utilise actuellement la version parallèle de cette méthode.
2. Pour réduire le temps de simulation, une autre solution a été imaginée, qui consiste à sauvegarder dans chaque case la distance jusqu'à toutes les ressources qui peuvent influencer son potentiel. Les ressources ne changeant pas de place, cette distance reste inchangée tout au long de la simulation. Dans ce cas, la propagation revient à calculer le potentiel *seulement* dans les cases où les agents ont besoin de connaître la valeur du potentiel. Cette solution est très rapide, cependant elle utilise beaucoup de mémoire. À titre d'exemple, un environnement 1024×1024 avec 2% de ressources peut avoir besoin de dizaines de Mo de mémoire.
3. Une autre solution pour implanter cet algorithme utilise le fait qu'une ressource *impose* son potentiel (il est le plus grand) sur un groupe de cases. Dans la figure 5, les trois ressources forment trois groupes de cases. Il est possible de spécifier pour chaque ressource son groupe, ou bien pour chaque case le groupe auquel elle appartient. Lors de la modification du potentiel d'une ressource, *seules* les cases voisines des frontières de son groupe sont réévaluées. Cette solution utilise moins d'espace que la solution précédente.

2.6. Entrées/sorties parallèles

Les fichiers nécessaires au système peuvent avoir de grandes tailles. La parallélisation des entrées/sorties peut décroître visiblement le temps d'exécution de l'application, et elle peut être acquise dans certains cas en travaillant parallèlement sur plusieurs fichiers. C'est une des raisons pour lesquelles il y a plusieurs fichiers de configuration et les informations sur le système sont écrites dans plusieurs fichiers [6].

2.7. Goulots d'étranglement

Généralement, les programmes ne sont pas complètement parallèles. La loi d'Amdahl [1] limite le gain obtenu par parallélisation par la fraction séquentielle de l'application. Les parties actuellement séquentielles dans la bibliothèque sont :

- L'initialisation de l'application est faite en séquentiel. Les régions d'exclusion mutuelle sont ainsi évitées. La parallélisation de l'initialisation est encore à l'étude.
- Pour garder la parallélisation transparente à l'utilisateur final, la fonction-utilisateur appelée à la fin de chaque cycle est exécutée en séquentiel. Cela peut ne pas être gênant si cette fonction représente une partie infime du temps d'exécution de l'application. En échange, les sauvegardes faites dans cette fonction ne sont que planifiées, et deviennent effectives après la fin de la fonction, lorsqu'elles seront faites en parallèle par le système [7].

2.8. Performances

La bibliothèque est encore au stade d'implantation et optimisation. Cependant, une application de type ParMASS a été écrite à l'aide de la bibliothèque : 800 agents se déplacent dans un environnement de 128×128 cases, avec 800 obstacles et 800 ressources. Elle a été exécutée sur un supercalculateur Origin2000

6. Partie non implantée encore
7. Cette parallélisation n'est pas encore implantée

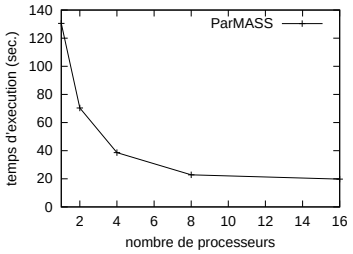

FIG. 6 – *Le temps d'exécution*

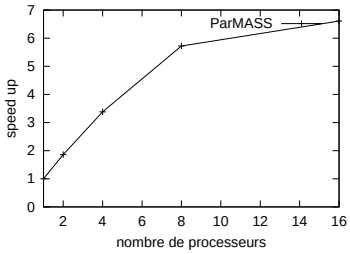

FIG. 7 – *Le speed-up*

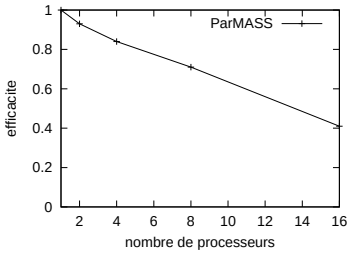

FIG. 8 – *L'efficacité*

de Silicon Graphics, avec 64 processeurs. L'option de compilation optimisée utilisée par le compilateur natif (MIPSPro) est `-Ofast=ip27` (optimisation standard maximum) et les tests ont été faits en mode multi-utilisateur. Les performances à l'exécution sont présentées dans les figures 6, 7 et 8. Le temps utilisé est le temps passé entre le début et la fin du programme. Les entrées/sorties sont négligeables.

3. Conclusion et perspectives

Cet article a présenté les spécifications actuelles d'une bibliothèque parallèle facilitant l'écriture d'une catégorie de systèmes multi-agent, dans laquelle le comportement des agents (réactif et/ou cognitif) peut utiliser des réseaux de neurones.

La bibliothèque est en cours de développement. Il reste à détailler certaines spécifications (comme la vision, la génétique des agents et la communication entre les agents) et à les évaluer, ainsi qu'à implanter ces fonctionnalités et la rendre plus performante.

L'avantage d'utiliser cette bibliothèque est de pouvoir créer rapidement des applications, de rendre le parallélisme à l'utilisation quasi-transparent et d'avoir de bonnes performances à l'exécution. Les utilisateurs de cette bibliothèque étant principalement les chercheurs en intelligence artificielle, dont souvent les programmes ont comme exigence principale l'implantation et l'exécution rapides, on peut espérer que cette bibliothèque, par sa simplicité d'utilisation et les fonctionnalités qu'elle fournit, sera un pas en avant dans l'expérimentation des systèmes multi-agent à composantes connexionnistes.

Remerciements

Ce travail est financé en partie par la *Région Lorraine*. Les expérimentations ont été faites sur le supercalculateur du *Centre Charles Hermite*.

Bibliographie

1. G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.
2. L. Bouton. Parmass : un simulateur de systèmes multi-agents parallèle. Master's thesis, Université de Nancy-I, Supélec, LORIA, June 1998.
3. T. Dagaeff, F. Chantemargue, and B. Hirsbrunner. Emergence-based cooperation in a multi-agent system. In *Proceedings of the Second European Conference on Cognitive Science*, pages 91–96, Manchester, U.K., 1997.
4. J. Fier. Performance tuning optimization for Origin2000 and Onyx. Available at `http://techpubs.sgi.com/library/manuals/3000/007-3511-001/html/O2000Tuning.0.html`.
5. L. Gasser. Agents and concurrent objects. *IEEE Concurrency*, 6(4):74–77, 81, Oct.-Dec. 1998. Interview taken by Jean-Pierre Briot.
6. N. Jennings and M. Wooldridge. Applications of intelligent agents. In N. Jennings and M. Wooldridge, editors, *Agent Technology. Foundations, Applications, and Markets*. Springer, 1998.
7. N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117:277–296, 2000.
8. L. Kipp. Application du parallélisme aux systèmes multi-agents. Master's thesis, Université de Nancy-I, Supélec, CRIN-LORIA, Sept. 1997.
9. B. Lester. *The Art of Parallel Programming*. Prentice Hall, 1993.
10. B. Nichols, D. Buttlar, and J. P. Farrel. *Pthreads Programming*. O'Reilly & Associates, Sept. 1996.